**The Dissertation Committee for James Carrell Holt certifies that this is the approved version of the following dissertation:**

# Evaluation of Dynamic Properties of Software Architectures Using Software Architecture Execution

**Committee:**

K. Suzanne Barber, Supervisor

Anthony P. Ambler

Don Batory

James C. Browne

Dewayne E. Perry

# Evaluation of Dynamic Properties of Software Architectures Using Software Architecture Execution

**by**

**James Carrell Holt, B.S.C.S.; M.S.C.S.**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

**May, 2003**

# Dedication

To Ofie, Aimee, and Chris.

So ends a twenty year journey begun at the University of Texas in 1983, passing full-circle through St. Edward's University and Southwest Texas State University, finally to return here where it all began, where I feel I belong. One person has been at my side for the duration: my wife Ofelia. Without her unfailing support and understanding I would not be writing this today. There are no words to express my gratitude for that.

Along the way we were joined by our children Aimee and Chris. The children haven't always understood what I was spending all of my supposedly free time doing, but they never complained or made me feel neglectful. My children, you have been a joy and inspiration to me, and I am thankful that you have helped me make it to this point. And to the rest of my family (especially to all of my favorites - you know who you are!) thank you too for everything you have done to sustain me. I love you all very much.

# Acknowledgements

Dr. Barber took a chance on me. I walked into her lab one day and felt as if I had stepped into a vortex. The amount of activity and energy was astounding. Yet, she stopped what she was doing, heard me out, and decided that day to give me a chance. Thereafter Dr. Barber treated me as a full member of the lab. I know this was risky, given that I was a non-traditional, part-time student. I am not sure any other faculty member would have been quite as willing to take that chance. I am very grateful to Dr. Barber for taking that chance on me, and have enjoyed every minute of working with her on this research. Dr. Barber, you have my everlasting respect, admiration, and gratitude. Thank you for everything, it has been an honor.

I want to thank the members of my Dissertation Committee, Dr. Ambler, Dr. Batory, Dr. Browne, and Dr. Perry, for your help and guidance. I know you are all very busy, and I appreciate your willingness to participate on the committee and dedicate some of your time to this process.

Working with the LIPS lab has been a wonderful experience for me. I only regret that I was not able to fully immerse myself in the lab, but then I have never done anything the 'normal' way. Still, everyone in the lab always treated me as a full member, even if they had not seen me for many weeks at a stretch. I truly enjoyed working with the SEPA team: Tom, Anita, Austin, Steve, Sutirtha, and Darla. Especially the collaboration with Tom Graser could not have been any more rewarding. I could not have pulled this off without Tom's help.

**Evaluation of Dynamic Properties of Software Architectures Using Software Architecture Execution**

Publication No._____

James Carrell Holt, Ph.D.

The University of Texas at Austin, 2003

Supervisor: K. Suzanne Barber

This dissertation shows that an integration of software architecture execution techniques is capable of evaluating multiple dynamic properties of requirements early and iteratively in the software development lifecycle. Contributions include a process and supporting tool for dynamic property evaluations, experimental results investigating the approach, and a case study involving an industrial software development project. Results show that the techniques developed in this dissertation can assist stakeholders in early detection and correction of requirements errors, and can provide rationale for decision-making associated with requirements trade-offs and evolution. Furthermore, the case study illustrates that the evaluation process and tool can be successfully employed to allow stakeholders who are not experts in software architecture execution to perform and analyze results of early dynamic property evaluations.

# Table of Contents

# List of Tables

# List of Figures

# CHAPTER 1 INTRODUCTION

This dissertation offers an integration of techniques capable of evaluating multiple dynamic properties of software architectures early in the software development lifecycle, with continuing iterative evaluations as the system evolves.

Dynamic properties of requirements are properties that are manifested during system execution. These properties include *correctness*, *performance*, and *reliability*. This research examines whether early and iterative dynamic property evaluations can have a substantial positive impact on initial requirements specification and subsequent requirements evolution.

The research approach involved representing requirements using software architecture models and evaluating those models using architecture execution techniques embodied in an evaluation tool called Arcade. Arcade integrates a number of techniques for evaluating dynamic properties, including *model checking*, *discrete event simulation*, and *graph algorithms*. These techniques are collectively referred to as software architecture execution techniques. Arcade was used to perform experimentation in support of the research hypothesis, and was applied in conjunction with an industrial software development project to determine the effectiveness of the approach.

It is important to note that early property evaluations of requirements cannot provide *quantitative* measures of ultimate system properties [19]. There are far too many factors encountered later in the software lifecyle that can affect

properties of the implemented system. Nevertheless, early property evaluations can provide a *qualitative* view of system properties. Results presented in this dissertation indicate that qualitative information from early dynamic property evaluations can serve several valuable purposes, including: (1) aiding in detection and correction of requirements errors, (2) aiding in decision-making associated with requirements evolution, and (3) providing guidance to system implementers.

The following sections elaborate on the research hypothesis and associated research questions, the research motivation, the research contributions, and the organization of this dissertation.

## 1.1 HYPOTHESIS AND RESEARCH QUESTIONS

The research presented in this dissertation was performed to investigate the following hypothesis:

> **Software Architecture Execution will allow for a systematic, automated evaluation of non-static (dynamic) architectural properties to assess dynamic properties in isolation, dependencies between dynamic properties, impact of early analysis and design decisions including the architectural derivation process and architecture specification content.**

An associated set of research questions was defined to create a framework for research and experimentation in support of the hypothesis. These research questions are listed below. Each research question is discussed in more detail in the following sections.

Research Question #1: *Given artifacts generated during the analysis and design phases of a software engineering process, what artifacts are required to support software architecture execution?*

Research Question #2: *What dynamic properties and property dependencies can be evaluated by executing the specification of software architectures?*

Research Question #3: *What is the systematic process and related techniques necessary to execute and evaluate a software architecture specification?*

Research Question #4: *Given decisions during the analysis and design phases of a software engineering process, which decisions can prove to impact the dynamic properties under software architecture execution evaluations?*

Research Question #5: *Can the rapid feedback from software architecture executions evaluations serve to influence analysis and design decisions in an iterative software engineering process?*

### 1.1.1 Research Question #1

*Given artifacts generated during the analysis and design phases of a software engineering process, what artifacts are required to support software architecture execution?*

As a prerequisite to supporting early evaluation of dynamic properties (e.g., during early analysis and design activities), it is important to establish what artifacts are available in these early phases and what information these artifacts contain that can support dynamic property evaluation. A given artifact is a candidate for evaluation if it contains information that contributes to the dynamic properties of the architecture. Dynamic properties are affected by functional requirements of a system, static structural aspects of a system, and non-functional constraints placed on the system. Consequently, artifacts that contain these types of information are good candidates for dynamic property evaluation.

### 1.1.2 Research Question #2

*What dynamic properties and property dependencies can be evaluated by executing the specification of software architectures?*

Off-the-shelf or other well-known tools and techniques for software architecture execution are capable of evaluating a number of dynamic properties. However, the goal of evaluating dynamic property evaluations early (e.g., before detailed design artifacts have been created) imposes some limitations on which

dynamic properties can be evaluated. Furthermore, individual dynamic properties are often evaluated in isolation rather than in concert. Therefore, the objectives of this research question are: (1) to determine what dynamic properties can be evaluated during requirements modeling, and (2) to investigate the benefits of evaluating a number of dynamic properties in an integrated tool.

### 1.1.3 Research Question #3

*What is the systematic process and related techniques necessary to execute and evaluate a software architecture specification?*

This research question focuses on understanding the tasks required to set up and perform dynamic property evaluations using software architecture execution. The objective is to establish a repeatable process for dynamic property evaluation using software architecture execution. This question also includes consideration of the practicality of employing the proposed software architecture execution techniques.

### 1.1.4 Research Question #4

*Given decisions during the analysis and design phases of a software engineering process, which decisions can prove to impact the dynamic properties under software architecture execution evaluations?*

The objective of this research question is to understand how architectural decisions affect dynamic properties. Two main classes of decisions are considered: (1) decisions associated with the process of structuring requirements into software architecture models, and (2) decisions associated with requirements evolution. This question was considered through experimentation covered in Chapter CHAPTER 5.

### 1.1.5 Research Question #5

*Can the rapid feedback from software architecture execution evaluations serve to influence analysis and design decisions in an iterative software engineering process?*

The objective of this research question is to investigate the efficacy of employing the results of dynamic property evaluations to aid in (1) early requirements acquisition and specification activities, and (2) subsequent requirements evolution activities. In support of this question, experimentation was performed to determine whether the software architecture execution evaluation approach was effective towards enabling rapid feedback from

evaluations, and whether such feedback can aid in detecting requirements errors and managing requirements evolution. Further examination of this question was performed through application of the approach to an industrial software development project.

The following section discusses the research motivation for this dissertation.

## 1.2 RESEARCH MOTIVATION

A fundamental goal of software engineering is to minimize the costs associated with developing and maintaining software systems. While software costs stem from factors encountered throughout the software lifecycle, it is known that the cost of addressing requirements errors increases as a function of how late in the software lifecycle they are addressed [19, 56, 104]. Therefore, it is desirable to evaluate requirements early in the lifecycle, when it is less costly to correct errors.

A number of researchers have demonstrated that dynamic property evaluations can be used to assist in detecting requirements errors [58, 102]. Most of that research has been targeted at evaluating individual dynamic properties using a static requirements set. A logical extension of that research is to investigate the evaluation of multiple dynamic properties as early in the requirements acquisition and modeling process as possible, when requirements are undergoing rapid changes. Therefore, a major research component of this

dissertation is evaluating multiple dynamic properties concurrent with requirements acquisition and specification.

While detection of errors during requirements acquisition and modeling is critical, requirements will likely continue to evolve over the life of a system [60]. Assessing the impact of requirements evolution as early as possible can aid in managing the costs associated with changes by helping stakeholders make decisions prior to implementation, and by subsequently providing system implementers with useful information regarding critical elements of a system. These benefits have been demonstrated by researchers studying how dynamic property evaluations of design-level software architectures can assist stakeholders in making decisions (e.g., choosing between architectural alternatives) [67, 70]. The research in this dissertation builds upon that experience by exploring how dynamic property evaluations can be employed earlier during requirements acquisition and modeling, and how dynamic property evaluation can continue to be employed in an iterative fashion as requirements evolve.

A key enabler for early and iterative dynamic property evaluations is the ability to perform evaluations using partial requirements models. This ability allows evaluation to proceed concurrently with requirements acquisition and modeling. Furthermore, as a requirements model becomes more complete over time (or as requirements evolve) the ability to perform evaluations of partial models can enable reuse of prior evaluation results for stable subsets of requirements [22]. Thus, reuse allows new evaluations to be focused at areas of change, helping to reduce the level of effort and complexity associated with

evaluation. Some research has been performed to understand the feasibility of using partial models with individual software architecture execution techniques [3, 28, 34, 35, 58, 102], but the focus has been on mitigating the complexity of performing evaluations. This dissertation extends that work by investigating how partial models can (1) enable evaluation of multiple dynamic properties early in the lifecycle, and (2) allow reuse of evaluation results as requirements mature and evolve.

Several feasible software architecture evaluation techniques have been developed [27, 34, 35, 61, 80, 82, 84, 85, 89, 105]. However, no single technique is suitable for evaluating the entire set of correctness, performance, and reliability properties examined in this research. The issues associated with integrating a number of software architecture execution techniques into a single tool are largely unexplored. The main issues involve the practicality of employing the techniques (e.g., the difficulty of using techniques, and the capacity of techniques to scale to real-world problems). While practicality issues associated with individual techniques have been examined in prior research [28, 34, 35, 57, 106], this dissertation explores practicality issues associated with integrating multiple approaches and making them available to stakeholders who are not experts in particular evaluation techniques.

The following section discusses how software architecture models can provide a framework to support early, iterative dynamic property evaluation using

partial models. This is followed by a discussion of the practicality issues associated with software architecture execution techniques.

### 1.2.1 Software Architecture Models for Requirements

While natural language provides a flexible representation for expressing requirements, it proves to be far too unstructured for effective evaluation [62]. Formal and semi-formal approaches offer greater structure to enable such evaluations. Semi-formal techniques involve diagram and tabular techniques that present information in structured form, whereas formal techniques leverage mathematics, logic, or algebra [110]. As an extension to formal requirements modeling techniques, software architectures are increasingly being investigated as frameworks for representing and evaluating requirements [10, 37, 45, 75, 100].

Observations from a previous empirical study indicate that different types of requirements (e.g., domain functionality, application look-and-feel, installation constraints) evolve at different rates [12, 13]. For example, accurately specified domain requirements (e.g., business process functionality, data and their relationships, timing between functions) are likely to evolve at a slower rate than technology-related non-functional requirements. In fact, domain requirements can support families of implemented systems [20, 21, 25], and may remain stable over the course of multiple technology cycles resulting in re-implementation of essentially the same domain functionality many times as non-functional requirements associated with specific technologies and installation sites evolve. In general, these research findings suggest that specifying requirements in

separate yet interrelated architecture representations encourages requirements reuse and facilitates requirements analysis, evolution, and refinement.

Recognizing that different requirements types evolve at different rates, the research in this dissertation was conducted in the context of the Systems Engineering Process Activities (SEPA) methodology and its associated SEPA 3D Architecture [10, 13]. The SEPA 3D Architecture is a model-based software requirements representation developed by researchers in the Laboratory for Intelligent Processes and Systems (LIPS) at the University of Texas at Austin. Distinguishing features of the SEPA methodology and the SEPA 3D Architecture include separation of requirements types among a set of interrelated software architecture models and support for concurrent software architecture derivation, evolution, and evaluation.

The SEPA 3D Architecture partitions requirements types among a set of interrelated software architecture models: the Domain Reference Architecture (DRA), the Application Architecture (AA), and the Implementation Architecture (IA). The DRA specifies *domain requirements* (e.g., business process functionality, data and their relationships, timing between functions), the AA specifies *non-functional requirements* (e.g., application look-and-feel, runtime performance requirements) and the IA specifies *installation requirements* (e.g., available site hardware platforms, middleware and communications software). Figure 1 depicts the relationships between these architectures and the contributions from all three types of requirements with respect to the demands of

customer sites and pre-existing or to-be-built applications under consideration. The SEPA 3D Architecture is described in more detail in Chapter 2.

The SEPA 3D Architecture supports partial requirements model evaluation by allowing partitioning in two dimensions: (1) partitioning across different requirements types, and (2) partitioning within a single requirements type. This explicit support for model partitioning can be leveraged to provide early requirements evaluation as well as on-going iterative evaluations.

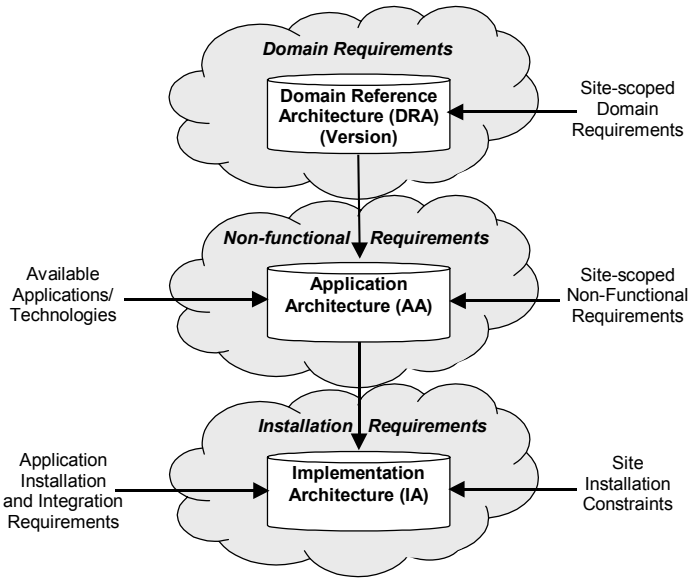


Figure 1 - DRA, AA, and IA Relationships

The first dimension is partitioning across requirements types (e.g., the DRA, AA, and IA models). For example, because the DRA is a highly abstract representation of functionality that is independent of any particular implementation, DRA evaluation can uncover issues associated with domain

requirements early in the requirements modeling and analysis process. In this example, evaluation of non-functional and site installation requirements modeled in the AA and IA can be deferred until a later time. Thus, DRA evaluation can proceed without requiring the AA and IA models.

The second dimension is partitioning within a single requirements type to create a partial model. For example, a partial model of a DRA might consist of domain requirements constrained to a subset of the domain functionality. A partial model can be used to focus evaluation on specific functionality of interest early during the requirements acquisition and modeling process, before domain requirements are completely modeled. The process can be repeated for different slices of domain functionality, without requiring the full set of domain requirements to be specified.

Partitioning also allows early evaluation results to be reused as new requirements are added or as requirements change. For example, if requirements were added or changed in the IA (e.g., the requirements changes affected site installation requirements), the results of prior DRA and AA evaluations would still be valid because there were no changes in domain requirements or non-functional requirements. Reuse makes the task of evaluating requirements changes simpler by reducing the amount of work involved in performing new evaluations, and by reducing the complexity of new evaluation results.

The following section discusses practicality issues associated with software architecture execution techniques.

### 1.2.2 Practicality of Software Architecture Execution Techniques

The *feasibility* of individual software architecture execution techniques has been a subject of research for some time, however *practicality* remains an open area of research that must be explored as a prerequisite to integrating multiple software architecture execution techniques and enabling widespread usage of the available tools and techniques by software practitioners [1, 7, 28, 34, 35, 38, 64, 69-71, 82, 85, 88, 90, 105, 106, 108, 111]. Two major open areas of research associated with the practicality of software architecture execution techniques are depicted in Figure 2: the expertise to conduct and analyze evaluations, and the capacity to process the magnitude of information resulting from evaluations.

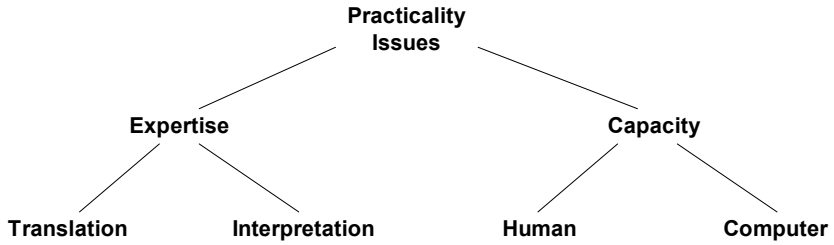


Figure 2 - Dynamic Property Evaluation Practicality Issues

Software architecture execution techniques require specialized expertise not typically held by software engineers [14]. Expertise is needed to translate a software architecture model into a format suitable for use with the selected evaluation technique. Expertise is also needed to interpret the results of evaluations and relate these results back to the software architecture model in a

meaningful way. Furthermore, the evaluation process is often iterative: as evaluations are completed, the software architecture model must be updated, and evaluation must be re-applied to examine the effects of updates. Even when expertise is available to a software engineer, the iterative tasks of translation and interpretation can be very time consuming, labor intensive, and error-prone when performed manually. Therefore, two barriers to successfully evaluating dynamic properties of software architecture models must be addressed: (1) the *translation barrier* (e.g., translation of a software architecture model to a form suitable for evaluation using off-the-shelf software architecture execution tools and techniques), and (2) the *interpretation barrier* (e.g., interpretation of the results of software architecture execution).

Several researchers have examined the translation barrier and the interpretation barrier [27, 82, 88, 115]. Much of this research has focused on evaluation of individual classes of dynamic properties centered around a single software architecture execution technique. When multiple techniques are integrated the effects of the two barriers are increased. This dissertation examines the two barriers in the context of evaluating multiple dynamic properties requiring multiple software architecture execution tools and techniques.

Capacity related issues of software architecture execution techniques include (1) the ability of humans to grasp large, complex software architecture models and their associated evaluation results, and (2) the scalability of software architecture execution techniques (for example memory and cpu requirements associated with the well-known state space explosion problem) [106]. Capacity

issues can be mitigated through partial model evaluations, but there is a need for more research addressing this topic. Therefore, this dissertation explores partial model evaluation techniques for mitigating both human and computer capacity issues.

The following section summarizes the research contributions of this dissertation.

## 1.3 RESEARCH CONTRIBUTION

This research provides experimental and applied evidence that early dynamic property evaluations of requirements can be accomplished by (1) using software architecture models to represent the requirements, and (2) leveraging software architecture execution techniques to perform evaluations. Furthermore, this dissertation demonstrates that a synthesized approach capable of concurrently evaluating multiple dynamic properties using multiple evaluation techniques can support early dynamic property evaluations, and can be employed in an iterative fashion as requirements evolve. In addition, partial requirements models are demonstrated to be an effective enabler for these tasks.

The research approach included development of a prototype dynamic property evaluation tool called Arcade. Arcade was used for experimentation in support of the research hypothesis, and an industrial case study.

Arcade is capable of evaluating a set of correctness, performance, and reliability properties of software architectures using a combination of software

architecture execution techniques: model checking and simulation for correctness evaluation, discrete event simulation for performance evaluation, and probabilistic graph models for reliability evaluation. This work has contributed a number of techniques to enable stakeholders who are not experts in software architecture execution to evaluate multiple dynamic properties of their requirements. These contributions include automated translation of software architecture specifications to various models required by the selected evaluation tools and techniques, and automated collection and presentation of evaluation results in terms that are intuitive to stakeholders.

The experimentation presented in this dissertation investigates dynamic property dependencies, the effects of various types of architectural decisions on dynamic properties, and the effectiveness of rapid feedback from dynamic property evaluations in aiding architectural decision-making. The experimental results indicate that the information contained in software architecture models is sufficient to support use of the selected techniques for early detection and correction of requirements errors, and can provide rationale for decision-making associated with requirements trade-offs and evolution.

Arcade was applied in a case study of an industrial software development project at Motorola. The empirical results obtained from the industrial project indicate that Arcade can enable early, iterative dynamic property evaluations in a real-world setting, and that these evaluations can have a positive effect on requirements acquisition and evolution. Furthermore, the case study validates that the automation provided by Arcade can effectively support non-expert

stakeholders in employing sophisticated dynamic property evaluation techniques for their purposes.

## 1.4 DISSERTATION ORGANIZATION

The remainder of the dissertation is organized as follows. Chapter 2 covers background material, and covers the SEPA 3D Architecture in more detail. Chapter 3 describes the research approach and the Arcade tool that was developed to support the research. Chapter 4 covers the eDesign case study. Chapter 5 presents the experimentation that was performed in support of the research hypothesis and research questions. Chapter 6 summarizes the research contributions of the dissertation and offers conclusions. The APPENDICES include a formal description of the Domain Reference Architecture (DRA) metamodel, algorithms employed in the Arcade prototype, and detailed experimental results associated with Chapter 5.

# CHAPTER 2 BACKGROUND

This chapter covers supporting material for the research in this dissertation. Two major topics are covered: (1) related research, and (2) the SEPA 3D Architecture.

## 2.1 RELATED RESEARCH

This section presents related research. The discussion is organized around the research questions that were introduced in Chapter 1. Where applicable, open research areas associated with the topic of this dissertation are identified.

### 2.1.1 Research Question #1

*Given artifacts generated during the analysis and design phases of a software engineering process, what artifacts are required to support software architecture execution?*

Many definitions of software architecture exist. The most enduring of these can be viewed as meta-definitions that encompass many of the more specific definitions. Perhaps one of the most widely cited and enduring definitions was proposed by Perry and Wolf [94]:

*Software Architecture = { Elements, Form, Rationale }*

Three classes of elements were identified in this definition: *processing elements*, *data elements*, and *connecting elements*. Processing elements transform data elements (and therefore have an associated behavior), while data elements contain information to be used by the system. Connecting elements are the glue that holds a system together.

The form of an architecture is a set of weighted properties and relationships. Properties are used to prescribe constraints on an individual element, while relationships prescribe constraints on how elements may interact, or how they may be structurally organized.

The rationale of an architecture captures the decisions made while creating the architecture. As the elements and form of the architecture are developed, the rationale will record the decisions that are made and their motivation.

In the context of this definition, most researchers investigating dynamic properties of software architectures have focused on processing elements and connecting elements, along with form. Artifacts which represent these aspects of a software architecture are supplemented with additional artifacts required to specify non-functional attributes of architectural elements.

Li et al., Kim et al., Duval et al., Tsai et al., Sharareh, et al., and Luckham et al. rely on software architecture artifacts expressed in Architecture Description Languages (ADLs) [71, 80, 85, 103, 106]. ADLs are languages that support the formal expression of the elements and form of an architecture. These formal models are typically executable and can be evaluated by a number of tools.

However, their formal nature often makes them difficult to use in industrial settings due to lack of expertise on the part of practitioners.

For this reason, much recent research has focused on using artifacts from the Unified Modeling Language (UML) to model software architectures [99]. These artifacts include class diagrams, message sequence diagrams, activity diagrams, and state charts [3, 36, 54, 91, 116]. The general trend in this research is to utilize the UML notations to describe the structural and behavioral aspects of a software architecture, and to use the UML extension mechanisms to augment the standard UML artifacts with non-functional information (for example performance and reliability attributes of architectural elements) to support dynamic property evaluations [40, 54, 91]. One such example is the work of Bose [27, 28]. This work extended the UML metamodel using stereotypes and constraints to support required architecture elements [98]. Similarly, Woodside and Petriu worked with early performance evaluations of software architecture requirements specified in Use Case Diagrams [96, 114]. This work focused on identifying attributes of these specifications that must be supplied in order to evaluate performance, and how those attributes should be supplied as extensions to UML artifacts.

Statechart diagrams (which have also become part of the UML standard) are frequently used as a basis for software architecture execution and dynamic property evaluation [46, 109]. To support these evaluations, statechart diagrams are typically annotated with information such as the probability of transitions, or the duration associated with a state. Such models can be transformed into a

number of representations that support software architecture evaluation including petri nets, and markov models.

Recognizing limitations in standards such as UML, several researchers have developed their own set of artifacts as the basis for their software architecture evaluations. For example, Lung et al. have defined a set of "architectural views" for their research [88]. These views include the Static View, the Dynamic View, the Map view, and the Resource View. The Static View includes artifacts such as static structure diagrams, object diagrams, and module diagrams. The Dynamic View includes state machine models, and petri net models. The Map view represents architectural styles employed, and architecture level patterns employed. The Resource View maps architectural elements onto hardware elements to allow for performance evaluations.

Li et al. have also defined their own set of architectural views [80, 82]. These views included the Logical View, the Development View, the Process View, and the Physical View. The Logical View consists of specifications of functions and services in the architecture. The Development View consists of static structural information. The Process View is used to allocate elements of the software architecture to processes. The Physical View is used to map processes onto hardware components. Kuusela et al. have defined a similar set of artifacts for their research on software architectures for Nokia [74].

In summary, a wide variety of analysis and design artifacts have been used for prior research in evaluating dynamic properties using software architecture execution. Regardless of the exact representation, the common information

modeled by these artifacts includes: (1) interface specifications for the architecture elements, (2) behavioral specifications of the architecture elements, (3) integration information based upon dependencies amongst architecture elements, (4) mapping information that allocates elements of the software architecture to computing environments, and (5) non-functional attributes of the architecture that affect dynamic properties such as performance and reliability.

As a result of this investigation, it was determined that the SEPA 3D Architecture artifacts could be used for the experimentation and case study in this dissertation. The SEPA 3D Architecture artifacts address all of the types of information described above using a set of related architecture models (Section 2.2). The Domain Reference Architecture (DRA) provides interface, behavioral, and integration specifications (Section 2.2.1). The Application Architecture (AA) provides mappings of elements in the DRA to Technology Solutions that implement the required interfaces and behaviors (Section 2.2.2). The Implementation Architecture (IA) provides mappings of Technology Solutions to computing environments (Section 2.2.3). Non-functional attributes such as service durations, frequencies of executions, and communication latencies are encompassed in these models as well. The benefits of using the SEPA 3D Architecture artifacts are described in more detail in Section 2.2.

### 2.1.2 Research Question #2

*What dynamic properties and property dependencies can be evaluated by executing the specification of software architectures?*

A general taxonomy for dynamic properties of software architectures that have been examined in recent literature is shown in Figure 3 [8, 19, 27, 28, 37, 44, 86, 101, 106, 111]. This taxonomy will be used throughout the remainder of the discussion of dynamic properties. Research associated with each class of property is described briefly in the following sections.

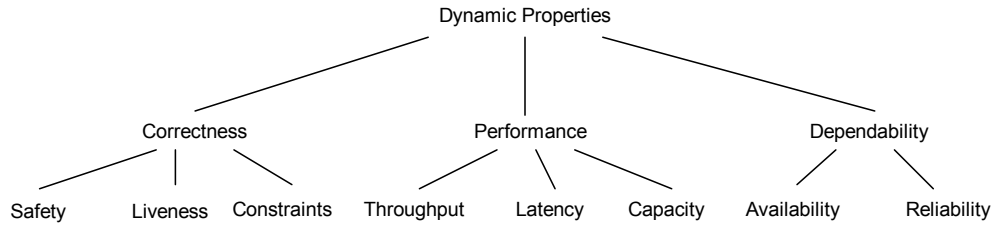


Figure 3 - Taxonomy of Dynamic Properties

#### *2.1.2.2 Correctness Properties*

Luckham has used Rapide simulations to evaluate deadlock conditions (e.g., safety) in architectural examples such as the Dining Philosophers problem [85]. These simulations are causal, and generate a partially ordered event set (poset) that captures the occurrence (or absence) of deadlock conditions. The constraint language of Rapide can be used to allow detection of other kinds of constraint violations in posets.

A number of researchers have used the SPIN model checker in association with software architecture evaluations. Tsai & Xu recently proved the ability to use SPIN/Promela [61] to detect deadlocks in several well-known software

architecture models [106]. The models included the "Gas-Station" problem, which is known to exhibit deadlock conditions [93]. Bose has also used SPIN/Promela to evaluate software architectures for safety and liveness properties, as well as constraints on ordering and causality of events [27, 28]. These techniques have been added to a prototype extended UML design environment for architectural design, and have been used to evaluate a software architecture for electronic commerce. Dias, et al., also report on a tool called Argus-I that incorporates SPIN for checking of properties of C2 style architectures [42].

Magee, Kramer, et al., have applied Compositional Reachability Analysis (CRA) to Labeled Transition Systems (LTS) specifications of software architectures [34, 35, 90]. The LTS specifications were supported by the Darwin ADL. The CRA technique has been shown to be suitable for evaluating safety and liveness properties, as well as for detection of deadlock conditions.

The collective research described above demonstrates that correctness properties are suitable for evaluation using software architecture execution techniques. However, although existing research has covered evaluation of correctness properties in isolation, there has been limited research covering evaluation of dependencies between correctness properties and other dynamic properties. In addition, the existing research has not focused on issues associated with evaluation of partial software architecture models (a necessary approach to allow evaluation to occur early and iteratively), nor has there been a focus on techniques to allow non-expert stakeholders to employ correctness evaluation

techniques. Therefore a contribution of this dissertation is investigation of early, iterative evaluation of correctness using partial models and employing techniques to enable non-experts to perform evaluations.

#### *2.1.2.3 Performance Properties*

Evaluation of performance properties at the system design level is a mature field. Software performance engineering methods and proven commercial design performance evaluation tools have been used for many industrial applications [30, 111]. Recently researchers have recognized that performance evaluation techniques that work effectively at the system design level can be extended to support higher levels of abstraction particularly by use of formal specification and compositional approaches [2, 29, 112]. Many software architecture researchers have taken cues from this more mature performance evaluation field.

Kuusela et al. have used colored petri nets with timed parameters in the Design/CPN toolset [66] to model and evaluate performance of telecommunications product families at the Nokia Research Center [75]. Specifically, they evaluated minimum, maximum, and average message delays in their architecture (e.g., latency).

Li has used simulations based on the SDL specification language to evaluate performance properties of architectures in the telecommunications domain [80]. The SDL language is a standard used by the telecommunications

industry. Specifically, Li used the commercial Telelogic SDL simulator to evaluate both throughput and latency of the architectures.

Wang and Chen et al., have studied the use of simulations of architectures to determine the impact of architectural style on performance [33, 108]. These simulations produced statistics to compare the processing times (e.g., latency) of the different architectural styles under various stochastic conditions related to their execution environment.

Lung et al. have used various simulation techniques to evaluate performance characteristics of architectures of telecommunications systems at Nortel's SEAL laboratory [88]. The properties they evaluated included bottlenecks (e.g., throughput issues), capacity, and load balancing (e.g., solutions to throughput issues). Their positive experience with this process has led them to propose automated methods for generating simulations from architecture descriptions.

Woodside, Petriu, et al. have published several papers describing their use of Layered Queuing Network (LQN) models for early performance evaluations [96, 113, 114]. Two main ideas emerge from their work: (1) the relation of software and hardware resources to a software architecture (and the resulting impact on performance), and (2) information required to evaluate performance in early software lifecycle phases. Their LQN models are advantageous over QNM models in supporting hierarchical composition of models.

The results of Kuusela et al., Li, Wang and Chen et al., Woodside, et. al., and Lung et al. show that performance properties of software architectures can be

evaluated using software architecture execution. Li has recognized certain dynamic property dependencies between performance and reliability properties. Wang, Chung, et al. have recognized dynamic property dependencies between performance, availability, and reliability. Both of these will be discussed further in Section 2.1.2.5.

None of this performance research has explored dependencies between performance and correctness properties, nor has there been significant research on techniques to enable such performance evaluations as early as possible in the software lifecycle by non-expert stakeholders. Therefore this dissertation contributes research regarding software architecture performance evaluations using incomplete requirements models (to allow for early evaluations), and using automated translation of these models to simulation input (to enable non-expert stakeholders).

#### 2.1.2.4 *Reliability Properties*

Li et al. have used simulations based on the SDL specification language to predict reliability of software architectures [81, 82]. With these simulations they were able to predict failure rates for elements of the architecture of a telecommunications system. This required a corresponding model of the system environment to model the effect of environmental failures as well as software failures. Wang and Chen et al. have also used simulations and markov models of architectures with different architectural styles to study availability and reliability

[33, 108, 109]. These properties were studied in terms of ability to recover, and recovery overhead.

While the use of simulation approaches has appeared more frequently in the literature [51], recent research has focused on additional approaches, including markov models, colored petri nets, slicing techniques, bayesian models, formal verification, and hybrid approaches combining simulation or analytical approaches with statistical techniques [39-41, 48, 51-53, 117, 118]. These techniques have been applied in various component-based reliability evaluation approaches. In general, this newer research recognizes the benefits of early evaluation of reliability properties, and while some researchers point out that reliability evaluations at the architectural stage have not been thoroughly researched, the benefits of early reliability evaluation have clearly been demonstrated.

Reliability properties have been studied less with respect to software architectures than performance and correctness properties, and may prove to be harder to evaluate and more subjective. The contribution of this dissertation is to extend research on evaluation of reliability properties of software architectures to include (1) evaluation of property dependencies between reliability properties and other dynamic properties, (2) reliability evaluation using partial models, and (3) enabling non-expert stakeholders to employ these techniques early and iteratively in the software lifecycle.

#### *2.1.2.5 Property Dependencies*

While there has been much discussion of the existence of dependencies between dynamic properties of software architectures [1, 8, 19, 24, 31, 37, 67, 70], few researchers have reported measured results associated with dependencies between dynamic properties.

Li et al. have reported the use of their simulations to perform tradeoff analysis between performance properties and reliability properties. This work was focused on determining the effects of introducing architectural elements specifically intended to enhance reliability properties [80, 82]. Wang and Chen et al. have also reported on their use of simulations of software architectures to determine tradeoffs between performance properties and availability properties [108], and between performance properties and reliability properties [33]. These researchers have shown that it is feasible to use software architecture execution to evaluate dynamic property dependencies.

The contribution of this dissertation is an investigation of detecting these dynamic property dependencies early in the software lifecycle. This investigation includes a statistical analysis of the number and types of dynamic property dependencies found in early software architecture models.

### 2.1.3 Research Question #3

*What is the systematic process and related techniques necessary to execute and evaluate a software architecture specification?*

A systematic process must address two aspects of software architecture execution: (1) translation of requirements specified in a software architecture model into a format suitable for dynamic property evaluation techniques, and (2) collection and presentation of evaluation results to stakeholders in intuitive formats. Ideally such a process would lend itself to automation. Furthermore, an important enabler for early evaluation is the ability to evaluate partial models using an incremental, iterative process (Section 1.2.2).

With regard to correctness evaluation, many researchers recognize the need to work with early, partial requirements models. For example, researchers applying model checking to requirements of a fault tolerant system devoted considerable effort towards mitigating the size of the state space that would result from a model representing their requirements [102]. The approach selected was to partition requirements based upon the class of fault that would occur during system execution if a requirement were violated. A model of the system design was created, and requirements were expressed in LTL formulae and then model-checked. This approach focused on the important topic of mitigating complexity issues associated with memory and cpu utilization, but did not address the issue of automatically creating the model to be checked from requirements specifications, or the issue of presenting results to non-expert stakeholders, nor was there a focus

on early evaluation in support of requirements evolution and reuse. Similarly, researchers using the Software Cost Reduction (SCR) approach acknowledged that presenting model checking results to users as logic formulae rather than in terms of the requirements models stakeholders were accustomed to remains an issue with their approach [58]. Other researchers have focused on compositional approaches to reduce complexity under model checking (e.g., partitioning by functional requirements encapsulation), but again the focus has not been on practicality issues associated with non-experts [34, 35, 119]. An approach to partitioning requirements to be model checked based upon scenarios has been presented, but this work did not address presentation of model checking results to stakeholders in non-expert terminology [28]. Gannod, et al., have developed a partially automated process for evaluating correctness properties of partial models associated with product line architectures [49]. Their approach involves model checking of models initially specified in an ADL. The translation between the ADL and the model checker was not fully automated, and expressing the specific properties to be checked required expert knowledge of the model checker. However, the resulting evaluations indicated some potential error conditions in a portion of the product line architecture, and when project engineers were presented with the errors it was verified that the errors were present in the actual system implementation, thus substantiating the value of early evaluations of partial models.

In the performance evaluation domain, researchers have applied various approaches including queuing network models (QNM) [4, 54, 95, 105], discrete

event simulation [80, 88], petri nets [91, 116], Software Performance Engineering (SPE) [111], and process algebras [23] for performance evaluation of software architectures.

Researchers working with QNM acknowledge a potential state explosion problem, and are seeking methods to reduce the model size based on MSC representations [3]. There is also a recognition that QNM based evaluation results must be translated into conclusions and recommendations for non-experts [95]. Researchers working with simulation approaches have recognized a need for automated translation of requirements into appropriate simulation models [80], and researchers working on process algebras have recognized that these formalisms need support for non-expert users (in this case via translation of architectural descriptions that integrate structural requirements with performance requirements into process algebra representations) [23]. Petri nets offer a formal model capable of performance evaluations, but open issues include automated translation from design and analysis notations to petri net models, and the possibility of automatically generating the execution scenarios that drive petri net models [48, 116].

While some of these research efforts recognize the need to support non-experts, and to allow partial model evaluation, most of these approaches treat software architectures as design-level artifacts rather than requirements representations. However, recent work by Petriu, et. al. using Layered Queuing Networks (LQN) describes sensitivity analysis and architecture comparison techniques necessary for early performance evaluation [96], and work by

Aquilani, et al. proposes an iterative process for performance evaluations of software architectures using architectures specified as Labeled Transition Systems (LTS) coupled with QNM techniques [4]. While these techniques were employed for performance evaluation, they are equally applicable to other types of dynamic properties, and similar techniques were employed in this dissertation for correctness, performance, and reliability.

Researchers working on reliability evaluation have used simulation [82], analytical methods [53], probabilistic models [117], and markov models [52]. While little emphasis has been placed on supporting non-experts in applying these techniques, sensitivity analysis techniques are frequently employed in this research [52, 118]. Sensitivity analysis can provide stakeholders with feedback regarding critical elements of their architectures without requiring that stakeholders interpret highly detailed evaluation results. As with performance evaluation, most of this work does not explicitly recognize architectural models as requirements models, and the topics of partitioning requirements to mitigate complexity, promote requirements reuse, or help with requirements evolution do not appear (although Yacoub uses scenarios as the basis for evaluation, there is no discussion of partitioning of the model [117]). Furthermore, none of this research addresses mitigating the need for expertise in the selected technique, or the ability to support early evaluation.

Some researchers have recognized the need for an integrated approach capable of evaluating multiple properties. The Argus-I tool was developed to support rapid feedback of both static and dynamic evaluations to stakeholders [42,

107]. The authors report that Argus-I incorporates both model checking and simulation to provide evaluations of correctness and performance properties. Similar to one of the major motivations for the research in this dissertation, the motivation for synthesizing multiple techniques in Argus-I is to provide an environment that makes these evaluation techniques more accessible to stakeholders. However, from what has been reported to date, it is not clear how Argus-I allows specification of correctness properties, nor is it clear how results of evaluations are presented to stakeholders.

Sharareh, et al., have used a domain-specific ADL as the basis for dynamic property evaluation and feedback [103]. Their research in support of Ericsson has proposed a detailed process for architecture specification, evaluation, and tradeoff analysis. The approach was demonstrated for tradeoffs between performance and modifiability. This approach was motivated by a desire to make the specification and evaluation techniques easy to use in an industrial setting, and they report that the use of the domain-specific ADL has partially achieved this goal by simplifying the specification process, however the representation of evaluation results to stakeholders has not been addressed.

In summary, a number of researchers have investigated topics related to processes and techniques for software architecture dynamic property evaluation. Researchers have investigated various aspects of automation, but there is a need for more research on automated translation of architectures to tool inputs, as well as formatting and presentation of results. The research in this dissertation investigates these areas.

### 2.1.4 Research Question #4

*Given decisions during the analysis and design phases of a software engineering process, which decisions can prove to impact the dynamic properties under software architecture execution evaluations?*

Many decisions made during the requirements gathering and specification process affect dynamic properties of systems. These decisions are related to functional requirements as well as non-functional requirements such as performance, and reliability [52, 113]. Examples include deciding the scope of functionality to be included in an architecture, deciding how to allocate functionality to architecture elements, deciding how to allocate software elements to hardware elements, deciding how to correct errors that are detected in requirements, and deciding required non-functional qualities.

Many researchers have reported that structural allocation decisions have a significant impact on dynamic properties of software architectures [4, 19, 41, 52, 91, 118], and there has been much evidence to support this in case studies. These decisions include the allocation of functionality to components, and the allocation of data attributes to components. Structural allocation decisions lead to decisions related to the interactions among architectural elements [79, 113]. This in turn determines dependencies between architectural elements. Such dependencies affect a number of dynamic properties including performance and reliability. Once structural decisions have been made, subsequent decisions that determine the allocation of software to hardware resources also affect dynamic

properties [40, 54, 111, 114]. These decisions include the distribution or co-location of components into hardware environments, as well as the specific performance and reliability attributes of computers and communication devices.

A new and largely unexplored class of architectural decisions are related to compositions of architectural patterns. A novel use of model checking to explore possible compositions of architectural patterns while looking for violations of data and event flows and service orderings required by domain functionality is reported in [73]. Future research on composition of systems from reusable components will likely leverage such approaches. This dissertation does not explore these kinds of decisions.

While the existing research has reported on the effects of allocation decisions (both allocation of functionality to software structure, and allocation of software to hardware), there are a number of other types of decisions that occur for software architectures. These decisions include the choice of how much functionality to include in an architecture, as well as the choices associated with how to correct errors that are detected in a software architecture. This dissertation contributes experimentation to examine the effects of these classes of decisions on dynamic properties.

### 2.1.5 Research Question #5

*Can the rapid feedback from software architecture execution evaluations serve to influence analysis and design decisions in an iterative software engineering process?*

Researchers investigating the use of architectural evaluations to influence analysis and design decisions have identified three important capabilities that such techniques require for effectiveness: (1) the ability to identify critical architectural elements, (2) the ability to compare alternative architectures based upon multiple attributes, and (3) automation in support of an iterative process [19, 25, 43].

Balsamo, et al. have proposed a mathematical architecture comparison technique for performance properties [6, 63]. This technique involves automated translation of architectures specified as Chemical Abstract Machines (CHAM) [64] into Queuing Network Models (QNM) and performing evaluations to determine limits of performance parameters for alternative architectures (for example the approach can compare alternative architectures based upon maximum throughput, etc.). This work supports both automated translation of an architecture to a suitable executable model and comparison of architectures. While the topic of identification of critical architectural elements has not been addressed directly, critical elements can be identified indirectly by varying model parameters to form alternate architectures that can be compared (whereas a direct approach would indicate where performance bottlenecks exist for a given architecture). In later research the CHAM models were replaced with Labeled

Transition Systems (LTS) [4], and eventually Message Sequence Charts (MSCs) in an effort to make this approach more tractable for software practitioners [3]. Other recent extensions to this work discuss the feedback and interpretation of results as part of an iterative evaluation process that can impact design choices [4]. The types of decisions examined were performance-related decisions such as the allocation of functionality to components, in which case the appropriate feedback in their methodology took the form of performance metrics for alternative architectures based on different structural allocations. Related work by Balsamo, et al., has explored combining the use of Stochastic Process Algebras (SPA) for software architecture specification with QNM evaluation of performance properties [5]. SPA specifications can be automatically checked for some forms of dynamic correctness (e.g., interface mismatches), thus the approach allows examination of correctness and performance properties. However, it was acknowledged that the SPA results were difficult for non-expert stakeholders to understand, and the ability to identify critical performance elements using QNM was not addressed. In general, while this thread of research has supported the notion of performance evaluation during early architecture phases, the authors have not applied these techniques in conjunction with industrial projects due to lack of tool support, and have therefore resorted to other less formal techniques when working with industrial projects [32].

Wang, et al., provide a case study illustrating tradeoff evaluation of alternative architectures based upon performance and availability properties [108]. This case study highlighted the benefits of presenting stakeholders with

information about multiple dynamic properties of candidate software architectures. While this research highlights the benefits of providing a rationale for selection of a particular architectural style, the process was not automated.

Bosch, et al., have developed an iterative process for architecture evaluation that is based upon using feedback results to produce an architecture design that meets non-functional quality goals [25]. The process involves producing an architectural specification, assessing its quality attributes using a number of techniques, and performing architectural transformations to improve quality attributes. The assessment and transformation phases are intended to be applied iteratively with feedback from the assessment phase governing the activities in the transformation phase. While the research does not specifically address automation of specific evaluation techniques or the transformation process, the approach has been validated in a number of industrial case studies [26].

Williams and Smith have extended their Software Performance Engineering (SPE) approach to create the Performance Assessment of Software Architectures (PASA) method [112]. This method focuses on performance engineering in isolation. However, it does propose a useful approach for iterative evaluation and feedback at the early architecture phase that could be generalized for other dynamic properties. Important features of PASA include evaluation of partial models based upon stakeholders' prioritized Use Cases, formatting and presentation of performance evaluation results to stakeholders, and recommendation of suggested architectural improvements for critical elements of

the architecture that have been identified by performance evaluations. This process is enacted by PASA experts who handle the tasks of translating an architecture into performance models, evaluating performance, and collecting and presenting results. While automation has not been a focus of the research, PASA has been applied in industrial situations successfully and validates the use of partial models for early architectural performance evaluations.

The SAAM method was developed to support qualitative architecture evaluation and rapid feedback of results for an individual architectural property such as modifiability [69]. Thus with SAAM, comparisons between architectures for an individual property can be performed, but analysis of dependencies between properties must be done by inspection using expert architects. The ATAM method was developed to provide an evaluation process for managing the tradeoffs between multiple architectural properties [70]. This is done by identification of architectural elements that are sensitive to more than one architectural property. As with PASA, both SAAM and ATAM require experts to enact the process on behalf of stakeholders. Recently the CBAM method has extended ATAM to include cost and benefit analysis as an aid in decision-making [68]. This is an important feature in an industrial environment, and in general the SAAM, ATAM, and CBAM methods have proved useful in helping the decision-making process in large complex projects and have been extended in a number of ways to support various domains [43, 77].

The research in this dissertation recognizes the contributions of other researchers in the area of architecture evaluation processes, and extends that work

by focusing on (1) early evaluation of multiple dynamic properties of partial models, (2) support for an iterative process that can re-use results of prior evaluations where applicable, and (3) enabling non-expert stakeholders to take advantage of dynamic property evaluation techniques to aid in decision-making.

## 2.2 THE SEPA 3D ARCHITECTURE

The SEPA 3D Architecture was introduced as a model-based requirements representation in Section 1.2.1. Requirements types represented by the SEPA 3D Architecture include: (1) domain requirements (e.g., business process functionality, data and their relationships, timing between functions), (2) application and system-wide non-functional requirements (e.g., application look-and-feel, runtime performance requirements), and (3) application integration and installation constraints/requirements (e.g., available site hardware platforms, middleware and communications software). These three distinct requirements types form natural boundaries between the architecture models comprising the SEPA 3D Architecture. Therefore, the SEPA Domain Reference Architecture (DRA) specifies domain requirements, while non-functional and installation requirements dictate design and implementation related concerns that drive the subsequent specification of Application and Implementation Architectures, respectively (AA and IA).

A high level static structure diagram of the SEPA 3D Architecture is shown in Figure 4. This diagram shows the relationships between the three architecture models of the SEPA 3D Architecture as well as main elements of the

models. The following sections discuss each of the three architectures in more detail.

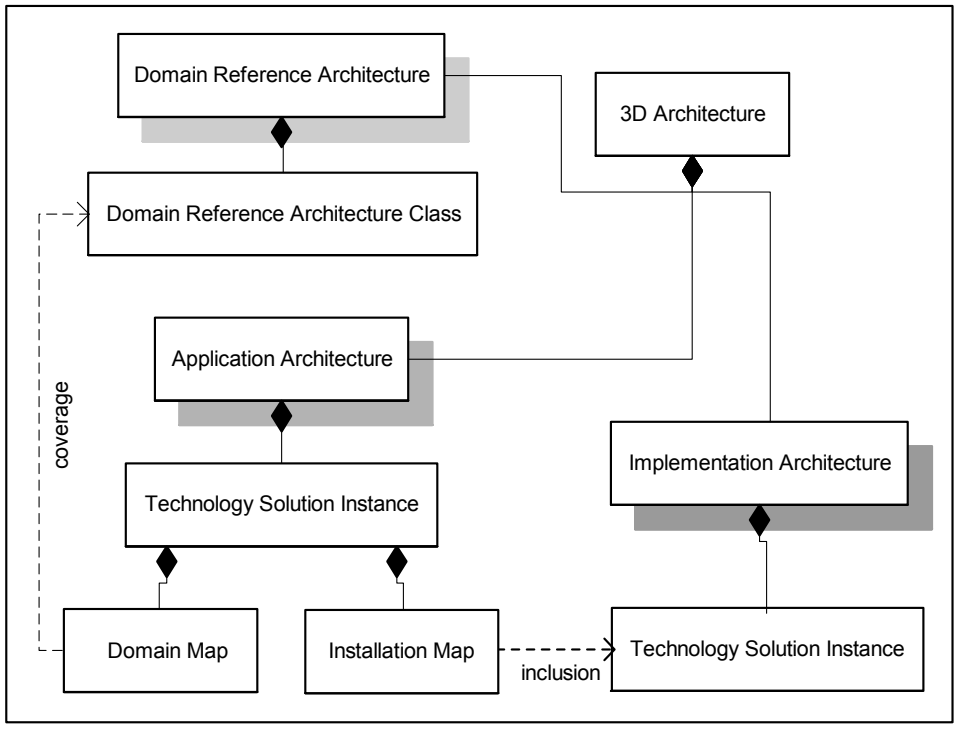


Figure 4 - The SEPA 3D Architecture

### 2.2.1 The Domain Reference Architecture

The Domain Reference Architecture (DRA) captures domain requirements, including business process functionality, data and their relationships, and timing between functions [16]. The DRA is composed of Domain Reference Architecture Classes (DRACs), each of which specifies some portion of domain data and functionality. These classes and their relationships

become a reusable blueprint that guides development efforts in terms of (1) the functional, data, and timing (i.e., ordering of functions) requirements to be satisfied, and (2) prescribed architectural structure specifying collections of and dependencies between (i.e. data or timing) system functionality. Each time the blueprint is reused for a new system development effort, DRACs may be instantiated by different applications (i.e., implementation solutions).

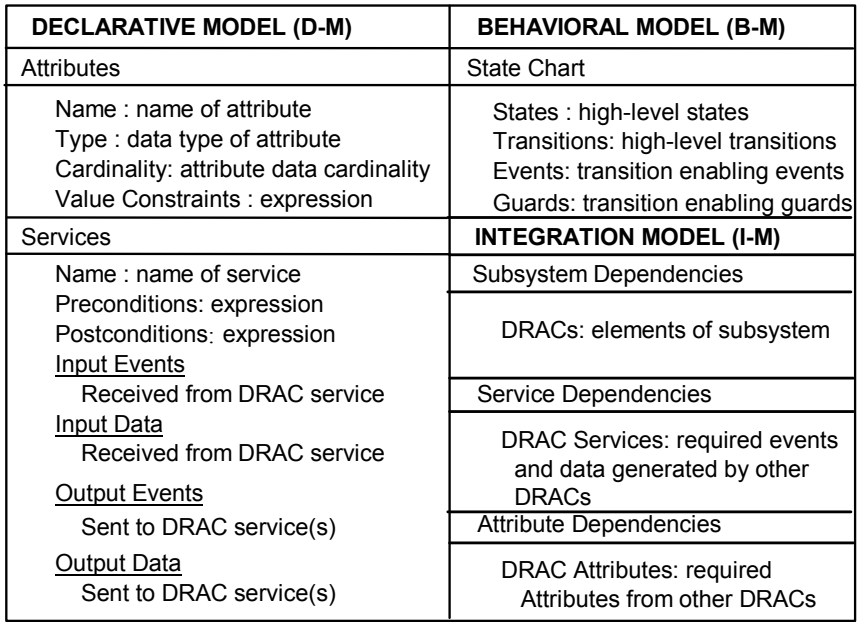


Figure 5 - Domain Reference Architecture Class Metamodel

The functionality and data allocated to a DRAC and the relationships between a DRAC and other DRACs are represented in the metamodel shown in Figure 5: the Declarative Model (D-M) defines the attributes (i.e. data and events) and services (i.e. functionality) that should be offered by an instance of the DRAC specification; the Behavioral Model (B-M) describes the behavior expected from

an instance of the DRAC through a high-level state chart; and the Integration Model (I-M) defines the constraints and dependencies between DRAC instances resulting from the distribution of dependent domain functions across DRACs. These dependencies are an artifact of the input and output of data and events among DRAC services (i.e., domain Function1 receives EventX from domain Function2) and are described in first order logic expressions (predicates) capturing service pre- and post- conditions.

The implementation-independence of the DRA is the salient feature that enables it to prescribe multiple systems in a domain. DRACs are not intended to map to typical implementation classes found in an object-oriented program (i.e., a Java or C++ object). Rather, DRACs express the architect's recommendations for partitioning domain data and functionality across applications. In a large system designed to support a customer site, it is common for multiple applications to cooperate in providing necessary domain functionality. DRACs provide developers and systems integrators with a blueprint that dictates how responsibilities should be distributed across applications and how those applications should interact based on that distribution. APPENDIX A contains a formal definition of the DRA.

### 2.2.2 The Application Architecture

The Application Architecture (AA) provides a framework for satisfying application requirements, including, but not limited to, application look-and-feel and runtime reliability requirements. The AA is formed when Technology

Solution (TS) instances (e.g., Applications, solutions implemented, under development, or envisioned) are selected to fulfill services (i.e., functions) specified in the DRA. These instances represent existing, under-development, or envisioned system components capable of providing certain services and data specified in one or more DRACs. The relation of these services to their associated DRAC(s) is called a coverage relation (Figure 4).

TS instances are selected from the SEPA Technology Solutions Repository (TSR), and may be Hardware, Software, or Personnel. These instances can be selected if they provide coverage of required services specified in the DRA (e.g., they have a coverage relation with the required DRA services) and if they meet implementation-related constraints imposed on these services by Application Requirements elicited from stakeholders.

When a TS instance is selected to fulfill DRA services, it may impose further requirements that necessitate incorporation of additional TS instances into the system. For example, the choice of a Software instance to fulfill a set of DRA services may further require incorporation of an "Operating System" TS into the system. This dependency relationship to one or more supporting TS instances is referred to as an inclusion relation (Figure 4). While both the coverage and inclusion relations are associated with a TS instance retrieved from the TSR, only coverage relations are satisfied in the AA. Inclusion relations are resolved in the IA.

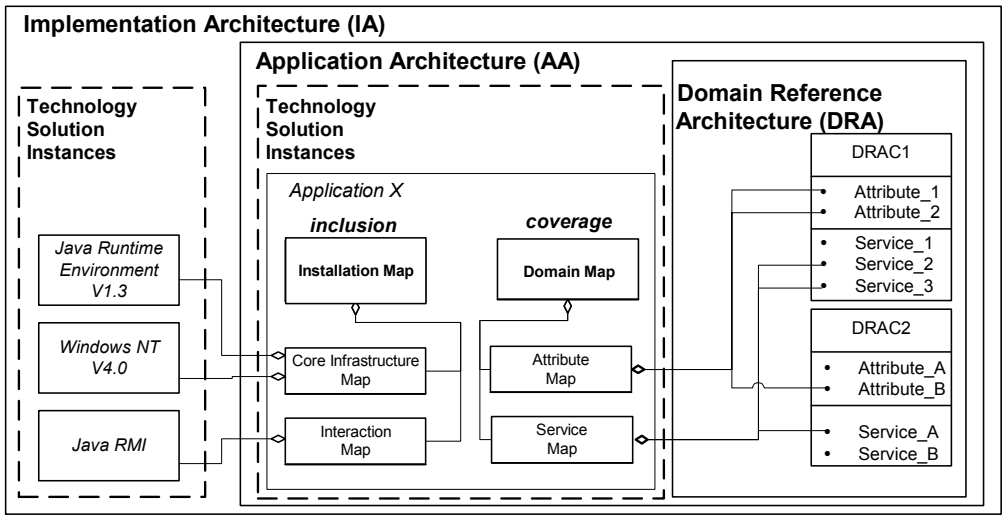


Figure 6 - Mapping Relations in the SEPA 3D Architecture

Figure 6 illustrates how a typical TS instance's coverage and inclusion relations are created using two mapping elements, a Domain Map and an Installation Map [13]. The Domain Map is instrumental in forming the AA, while the Installation Map is instrumental in defining the IA (discussed in the following section). A Domain Map is comprised of a set of Attribute Maps and Service Maps that define the coverage relation through links to the domain attributes (data) and services (functions) satisfied by a TS instance. In the example shown in Figure 6 "Application X" provides coverage of "Service_2" and "Service_3" from "DRAC1," and "Service_A" from "DRAC2." This illustrates that the coverage relation need not be one-to-one between TS instances and DRACs.

In the SEPA methodology, integrating applications when configuring a site involves both domain and implementation considerations, where

implementation considerations include non-functional requirements and installation constraints. The DRA expresses the domain integration considerations (ensuring I/O compatibility between DRAC services), allowing them to be reused independently of implementation considerations. From a domain perspective, dependencies between services (domain functions) result from the exchange of data or events between those services. When such services are allocated to DRACs, dependencies between DRACs result, and a technology providing the services in a given DRAC (or set of DRACs) inherits these dependencies. Figure 7 illustrates this type of domain dependency.

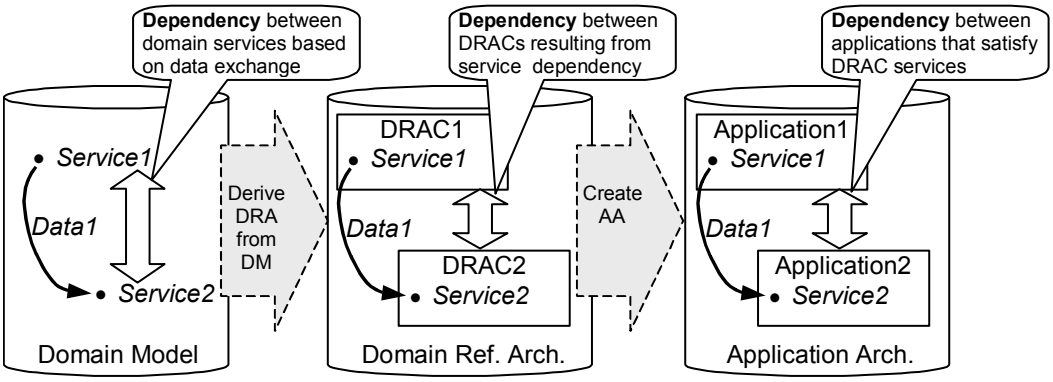


Figure 7 - Domain Dependencies Between the DRA, and AA

In Figure 7, domain functions “Service1” and “Service2” in the Domain Model have a dependency based on the exchange of “Data1”; “Service2” cannot execute until “Data1” is produced by “Service1”. In the Domain Reference Architecture derived from the Domain Model, Service1 and Service2 are allocated to DRACs DRAC1 and DRAC2, respectively. As a result, a

dependency arises between DRAC1 and DRAC2. When applications selected for an AA comply with the service groupings defined by DRACs in the DRA, DRAC dependencies become corresponding dependencies between applications. In Figure 7, applications "Application1" and "Application2" provide the services associated with "DRAC1" and "DRAC2", thereby inheriting their data dependency. Such a dependency would arise between any pair of applications adhering to the specifications defined by "DRAC1" and "DRAC2", and thus the dependency is inherent to the domain and implementation-independent.

### 2.2.3 The Implementation Architecture

The Implementation Architecture (IA) supports the satisfaction of site installation requirements, including constraints dictating site-specific hardware platforms, middleware, and communications software. To form the IA, additional TS instances are selected from the TSR to satisfy the inclusion relations of the AA (Figure 2, Figure 6). The inclusion relations are specified by one or more Installation Maps associated with a TS instance in the AA. The mappings from TS instances in the TSR to elements of a SEPA 3D Architecture are depicted in Figure 8.

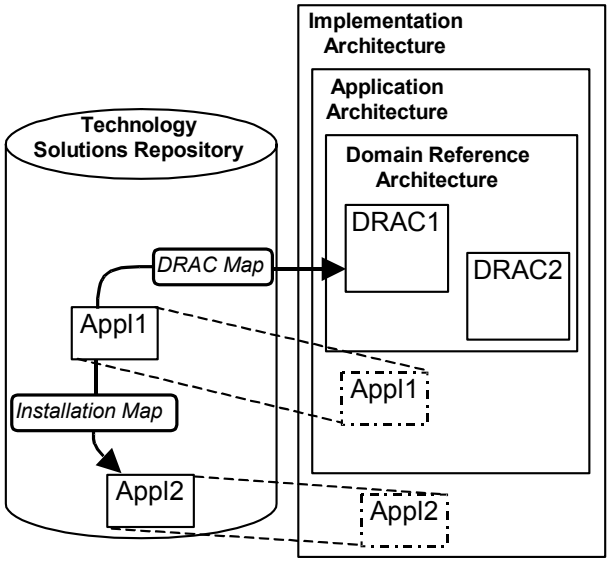


Figure 8 - TSR With Instances and Corresponding Maps

An Installation Map is comprised of a set of Core Infrastructure Maps and Interaction Maps. These maps define the inclusion relation through links to TS instances that a TS instance in the AA depends upon. The Core Infrastructure Map captures platform information for TS deployment (e.g., "Application X" requires an instance of a Runtime Environment and an Operating System). The Interaction Map captures information relevant to integrating an application into a system (e.g., "Application X" requires an instance of type Middleware to enable communication).

The example in Figure 6 illustrates the traceability of these inclusion relations. In this example "Application X" is linked through its Installation Map to specific TS instances "Java Runtime Environment v1.3," "Windows NT v4.0,"

and "Java RMI." Alternative IAs could specify completely different TS instances, as long as the constraints imposed by the inclusion relation for "Application X" are satisfied.

Application dependencies resulting from implementation considerations typically vary across installations, depending on the applications selected and the resources available at a given site. The IA in Figure 9 extends the Application Architecture in Figure 7 by considering the implementation requirements for a particular site. While Application1 and Application2 together provide the necessary domain services, Service1 and Service2, the applications were designed to run on different operating systems. Thus, while these applications satisfy all domain dependencies, an implementation conflict arises when identifying an acceptable operating system during site configuration.

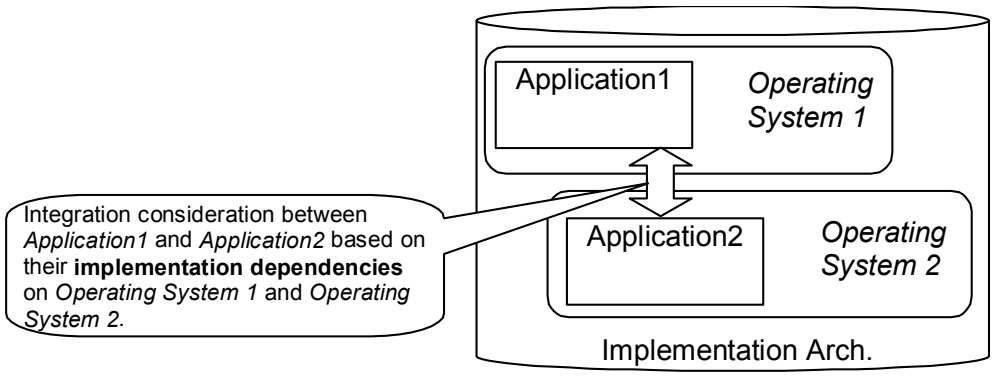


Figure 9 - Integration Considerations of Application Requirements

# CHAPTER 3 APPROACH

This research incorporated a multi-faceted approach designed to investigate the hypothesis and associated research questions. The approach included: (1) an extensive survey of published research on software architecture execution and dynamic properties, (2) development of integrated evaluation techniques embodied in a prototype evaluation tool, (3) experimentation in support of the hypothesis and research questions, and (4) an industrial case study employing the research for dynamic property evaluation.

This chapter begins by describing aspects of the research approach as they relate to each of the research questions. This is followed by details regarding the implementation of Arcade, the prototype dynamic property evaluation tool developed in support of this research. The eDesign case study is presented in Chapter CHAPTER 4, and the experiments are presented in Chapter CHAPTER 5.

## 3.1 APPROACH TO RESEARCH QUESTIONS

This section describes the research approach used for this dissertation. The discussion is organized by research question. Where appropriate, details of the literature review, experimentation, or case study are summarized.

### 3.1.1 Research Question #1

*Given artifacts generated during the analysis and design phases of a software engineering process, what artifacts are required to support software architecture execution?*

The approach to examine this research question was based upon an extensive literature survey. This survey determined that the literature contains an impressive amount of information regarding analysis and design artifacts required for the evaluation of dynamic properties using software architecture execution techniques. The details of the literature survey were covered in Chapter 2. The findings of the literature survey are summarized below.

The analysis artifacts used in prior software architecture execution research have been produced by a number of different analysis approaches and have had various representations [27, 28, 71, 80, 82, 87, 88, 106]. Regardless of the analysis approach and details of representation, researchers have consistently reported that software architecture execution requires several types of analysis artifacts, including: (1) interface specifications for the architecture elements, (2) behavioral specifications of the architecture elements, (3) integration information based upon dependencies amongst architecture elements, (4) mapping information that allocates elements of the software architecture to computing environments, and (5) non-functional attributes of the architecture that affect dynamic properties such as performance and reliability. All of these artifacts are found in the SEPA

3D Architecture. As a result of investigating this research question it was determined that the SEPA 3D Architecture provided the necessary artifacts to support the implementation and experimentation tasks for this dissertation.

### 3.1.2 Research Question #2

*What dynamic properties and property dependencies can be evaluated by executing the specification of software architectures?*

The approach for examining this research question was to (1) perform a literature survey to identify dynamic properties and dynamic property dependencies that have been studied in prior software architecture execution research, and (2) identify experimentation required to address significant open research areas. A brief summary of the literature survey is presented in the following paragraphs. The literature survey uncovered a need for further experimentation regarding dynamic property dependencies. The details of the literature surveyed in support of this research question are covered in the background material in Chapter 2. The experimentation is summarized below and covered in detail in Chapter CHAPTER 5.

Sufficient research results have been reported to support identification of a set of dynamic properties that can be evaluated using software architecture execution (Figure 10). From the larger set of properties that have been examined in previous work, a subset of properties was selected for this research (e.g., *correctness*, *performance*, and *reliability)*. The selection of this set of properties

was based upon (1) the importance of the properties in achieving high quality systems, and (2) the availability of existing software architecture execution tools and techniques for evaluating these properties.

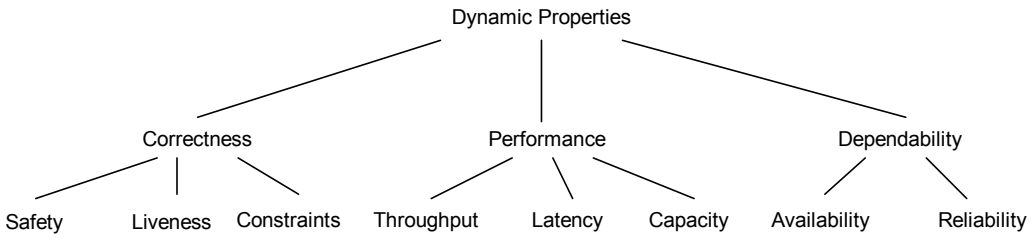


Figure 10 - Taxonomy of Dynamic Properties

While there has been considerable research on evaluating a variety of dynamic properties, the topic of evaluating dynamic property dependencies using software architecture execution techniques has not been widely represented in reported results. Therefore, the research in this dissertation includes experimentation to explore how dependencies between dynamic properties can be evaluated using software architecture execution. The experimental approach used to explore this question consisted of evaluating a number of different architecture models in which a specific dynamic property was emphasized by heuristics governing structural allocation of requirements (Section 3.1.5). Evaluation results were analyzed to determine that statistically significant correlations could be found between dynamic properties of the set of architectures.

### 3.1.3 Research Question #3

*What is the systematic process and related techniques necessary to execute and evaluate a software architecture specification?*

A systematic process requires well-defined steps that can be applied in a repeatable manner. For the purposes of this research, it is important that such an approach can (1) be employed early and iteratively, (2) allow stakeholders who are not experts in software architecture execution to perform evaluations, and (3) lend itself to automation. Therefore, the approach to this question was to define a process for dynamic property evaluation that leveraged the SEPA 3D Architecture in conjunction with a number of software architecture execution techniques. The resulting evaluation process was automated by the Arcade prototype, and subsequently applied in the context of the eDesign case study to determine how well the approach could accomplish the goals stated above. The Arcade dynamic property evaluation process is depicted in Figure 11.

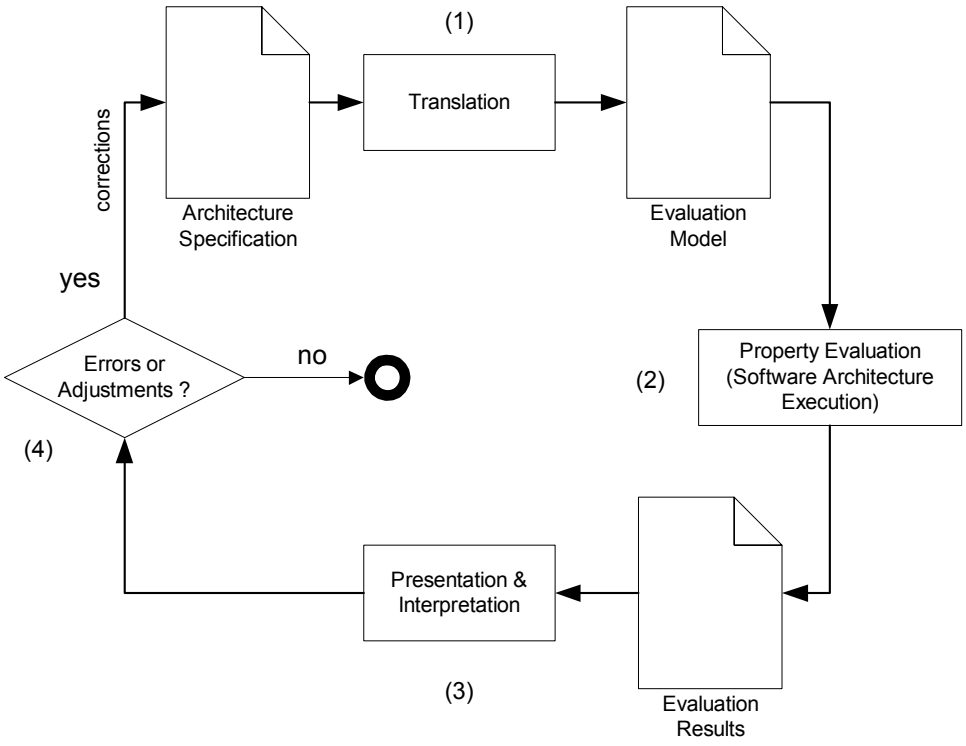


Figure 11 - Arcade Evaluation Process

The process begins with translation of an architecture specification into one or more models suitable for the software architecture execution techniques employed by Arcade (step 1 in Figure 11). Arcade performs these translations automatically on behalf of stakeholders, providing support for stakeholders who do not have expert knowledge on how to employ underlying evaluation techniques. Following translation, evaluation is automatically performed and evaluation results are produced in a number of formats specific to the techniques employed (step 2 in Figure 11). Arcade transforms these evaluation results into a number of graphical and textual representations designed to make the results of

evaluations easily understandable to Arcade users who are not experts in underlying evaluation techniques and representations (step 3 in Figure 11). Stakeholders can then analyze the results and determine if any corrections or adjustments are necessary based upon evaluation results (step 4 in Figure 11). If modifications are made to the architecture, the process can be repeated to determine the effects of modifications.

As requirements for a system evolve over time, it is likely that stakeholders will wish to evaluate the effect of requirements evolution on the dynamic properties of their architecture, or may wish to compare the dynamic properties of different architecture candidates relative to one another. The process for these kinds of evaluations is depicted in Figure 12. This process is an extension to the process for evaluating an individual architecture. First, a set of evaluation results are produced for each architecture version to be compared (step 1 in Figure 12). This is done using the individual architecture evaluation approach (Figure 11). Once a set of evaluation results has been produced, a number of comparison techniques are employed to rank the architectures in terms of individual properties, classes of properties, and overall (step 2 in Figure 12). These comparison techniques are described in Section 3.2.12.

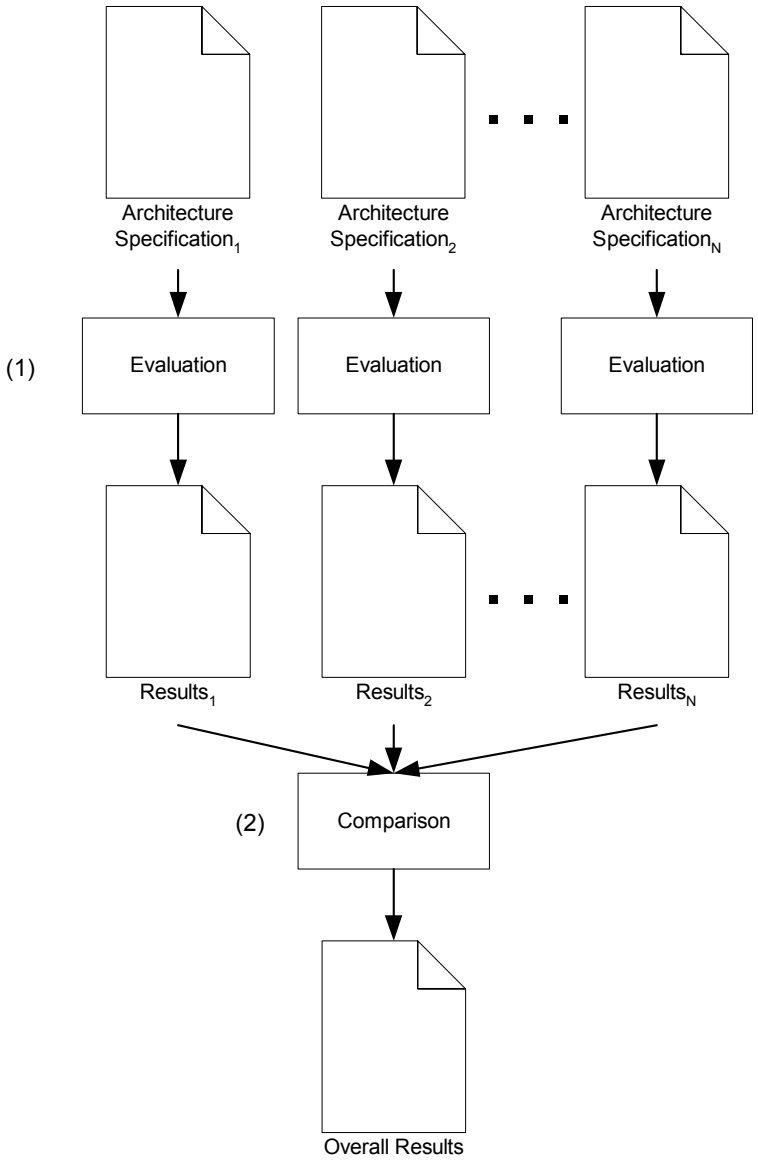


Figure 12 - Arcade Process for Comparing Architecture Versions

An important enabler for early evaluation is the ability to perform evaluations of partial models (Section 1.2). The Arcade approach leverages the SEPA 3D Architecture to support partial model evaluations using the model

partitioning dimensions supported in the SEPA 3D Architecture (Section 1.2.1). These dimensions are: (1) partitioning within architectures (for example a partial model of a DRA representing a slice of domain functionality), and (2) partitioning across architectures (e.g., the different requirements types modeled by the SEPA DRA, AA, and IA). The eDesign case study used Arcade's partial model evaluation process extensively (Chapter CHAPTER 4).

### 3.1.4 Research Question #4

*Given decisions during the analysis and design phases of a software engineering process, which decisions can prove to impact the dynamic properties under software architecture execution evaluations?*

This research question was examined through experimentation. Details of the experimental plan and results are covered in Section 5.8. Experiments were designed and executed to explore the effects of two classes of decisions on dynamic properties: (1) decisions associated with the process of structuring requirements into software architecture models, and (2) decisions associated with requirements evolution.

Structural decisions involve allocations of functionality, data, and events to components. Experiments evaluated a set of architectures with various structures for a fixed set of requirements obtained from the eDesign case study (Section 5.8.1). The dynamic properties of each of these architectures were compared using the Arcade comparison techniques described in Section 3.2.16.

The results of these comparisons indicate that the effects of structural decisions can be effectively communicated to stakeholders using the Arcade comparison approach.

In addition to structural decisions, two classes of *requirements evolution* decisions were considered in this research question: (1) within-architecture evolution decisions, and (2) across-architecture evolution decisions. Two experiments were defined for understanding the effects of decisions associated with requirements evolution.

The approach used for the within-architecture experiment was to evaluate a set of software architecture models created by incrementally adding requirements to a base set of requirements. These requirements were obtained from the eDesign case study. Results of this experiment indicate that the Arcade comparison process can effectively detect and communicate relative changes in dynamic properties when within-architecture evolution occurs.

The across-architecture experiment explored how the Arcade comparison process could evaluate and communicate the effects of (1) decisions associated with mapping multiple AA versions to a single DRA version, and (2) decisions associated with mapping multiple IA versions to that set of AA versions (Section 5.8.4). Results of this experiment indicate that distinct patterns of architectural evolution can be identified using the Arcade comparison approach, and these patterns can help stakeholders identify how their across-architecture decisions affected the dynamic properties of architectures.

### 3.1.5 Research Question #5

*Can the rapid feedback from software architecture execution evaluations serve to influence analysis and design decisions in an iterative software engineering process?*

A combination of implementation, experimentation, and the eDesign case study were performed to examine this question. The implementation task involved creating a feedback mechanism between Arcade and the Reference Architecture Representation Environment (RARE), a research tool designed to aid in the structural allocation process [55]. RARE facilitates the process of structuring domain requirements into a Domain Reference Architecture (DRA), a formal requirements model (APPENDIX A). RARE assists architects with DRA derivation using a heuristics-based approach that relies on static property metrics applied to a set of weighted quality goals such as maintainability, performance, and reliability.

Software architectures exhibit both static and dynamic properties (static properties include *modifiability*, *portability*, *reusability*, etc). In isolation, RARE can only calculate static property metrics for a software architecture, and thus is limited to the use of static property heuristics in the derivation process. The contribution of this research is to enhance RARE's derivation process by supplying dynamic property metrics to enable the use of dynamic property heuristics in conjunction with static property heuristics. Accordingly, a

mechanism was defined whereby Arcade evaluation results could be fed back to RARE during derivation [15]. This feedback mechanism is shown in Figure 13.

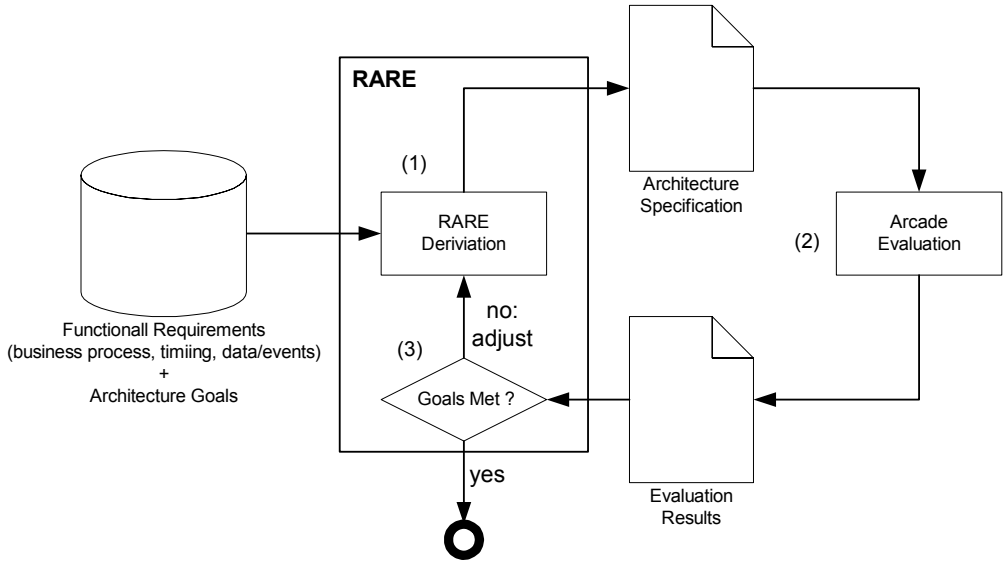


Figure 13 - Feedback Mechanism Between RARE and Arcade

The feedback process begins with RARE producing a DRA specification using a set of weighted static and dynamic property goals supplied by stakeholders (step 1 in Figure 13). Once an initial architecture specification is derived, its dynamic properties are evaluated using Arcade (step 2 in Figure 13) and the evaluation results are supplied back to RARE. RARE then uses the Arcade evaluation results to determine whether stakeholder goals have been met (step 3 in Figure 13). If the goals have not been met, RARE can use the knowledge gained from the Arcade evaluation results to apply additional

heuristics based on dynamic property metrics to derive another architecture version.

Experimentation was performed to determine whether feedback of dynamic property evaluation results from Arcade to RARE could positively influence decisions during the structural allocation process. The results of the Arcade to RARE feedback experiment indicate that the availability of feedback from Arcade's dynamic property evaluations can positively affect the structural allocation decisions made by RARE during derivation. Detailed results of this experimentation are presented in Chapter CHAPTER 5.

## 3.2 IMPLEMENTATION OF EARLY DYNAMIC PROPERTY EVALUATIONS

The goal of the Arcade implementation is to provide a systematic, automated approach for early dynamic property evaluation of requirements. To accomplish this goal, Arcade leverages the SEPA 3D Architecture in conjunction with a number of dynamic property evaluation techniques, including model checking, discrete event simulation, and probabilistic graph model algorithms. The details of the Arcade design are presented in the following sections.

### 3.2.1 Arcade Design

Arcade provides a software framework for early dynamic property evaluations. A number of capabilities are implemented in this framework:

**Capability 1:** *Mapping from SEPA artifacts to Executable Software Architecture Models*. Arcade translates the three SEPA 3D Architecture models to a single model using the Arcade metamodel (Section 3.2.2). The advantage of this approach is that fewer types of translations between architectures and evaluation tools are required. Arcade uses its metamodel as a common starting point for translation to other models as appropriate for software architecture execution tools.

**Capability 2:** *Setup and Control of Executable Software Architecture Model Parameters*. Examples of model parameters include mean service execution time, probability of usage profile execution requests, relative rates of execution, and communication channel latency. Allowing stakeholders to manipulate model parameters provides an opportunity to evaluate "what-if" scenarios.

**Capability 3:** *Software Architecture Execution and Collection of Results.* This capability is a key enabler for dynamic property evaluation and for the experimentation performed in this dissertation. Arcade automates dynamic property evaluations, and provides evaluation results in a number of formats that are designed to be intuitive for stakeholders. This automation enables stakeholders who are not experts in the sophisticated evaluation techniques employed by Arcade to benefit from the power of those evaluation techniques.

The Arcade framework is show in Figure 14. Major components of the framework include the Translation Engine, the Setup and Control Interface, the Execution Controller, the Execution Monitor, and the Presentation Interface.

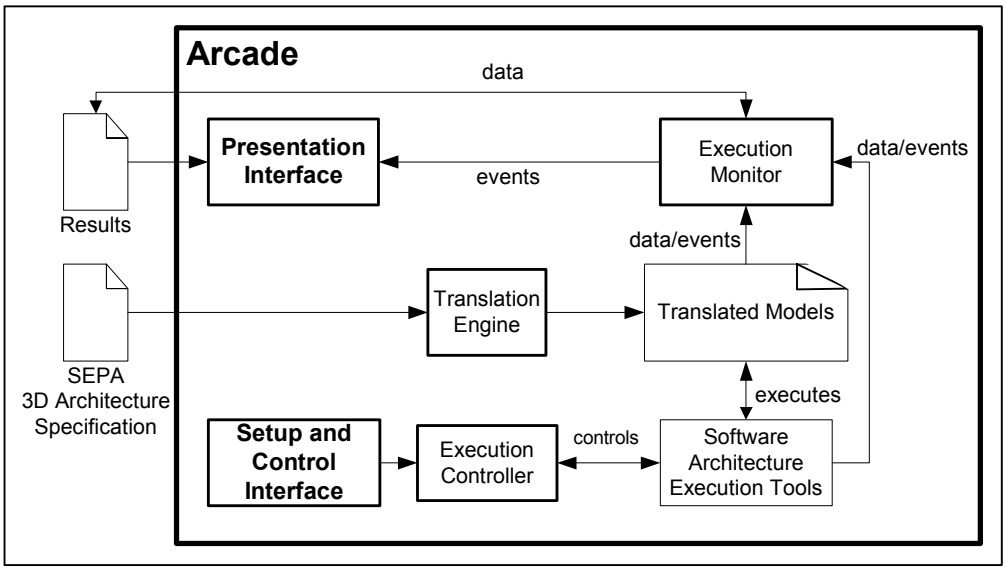


Figure 14 - The Arcade Framework

The Translation Engine provides all functionality associated with translating SEPA 3D Architecture models into Arcade models, and also provides translation from Arcade models to appropriate models for evaluation tools. The Setup and Control Interface provides the ability to adjust certain parameters of software architecture execution tools (specific parameters are described in sections covering Arcade evaluation approaches for their associated properties). The Execution Controller coordinates between Software Architecture Execution

Tools and the Arcade framework. The Execution Monitor captures results from software architecture execution tools. These results can be saved to data files or supplied to the Presentation Interface. The Presentation Interface provides a number of graphical and textual representations of the models under evaluation as well as evaluation results associated with those models.

The following sections describe the Arcade metamodel and how it is used in conjunction with the SEPA 3D Architecture and software architecture execution tools to provide the capabilities described above.

### 3.2.2 Arcade Metamodel

Figure 15 introduces the elements of the Arcade Architecture metamodel as an entity-relationship diagram. The Arcade metamodel is based upon a components and connectors architectural model [19], augmented to contain information related to computing hardware on which the architecture is deployed. Thus, an Arcade Architecture is composed of one or more components and one or more connectors. Each component is allocated to exactly one compute environment, and each connector is allocated to exactly one communication channel. A component owns zero or more attributes and provides one or more services. Each attribute of a component represents either a data item or an event. Services own zero or more parameters. Parameters are the means by which services (1) receive input data and input events, and (2) produce output data and generate output events. Services include a pre-condition and a post-condition that

specify the state of the architecture upon service initiation and following service completion, respectively. Pre-conditions and post-conditions are first order logic expressions composed of operators and operands. Operand instances may target one of the set {`expression, attribute, parameter, literal`}.

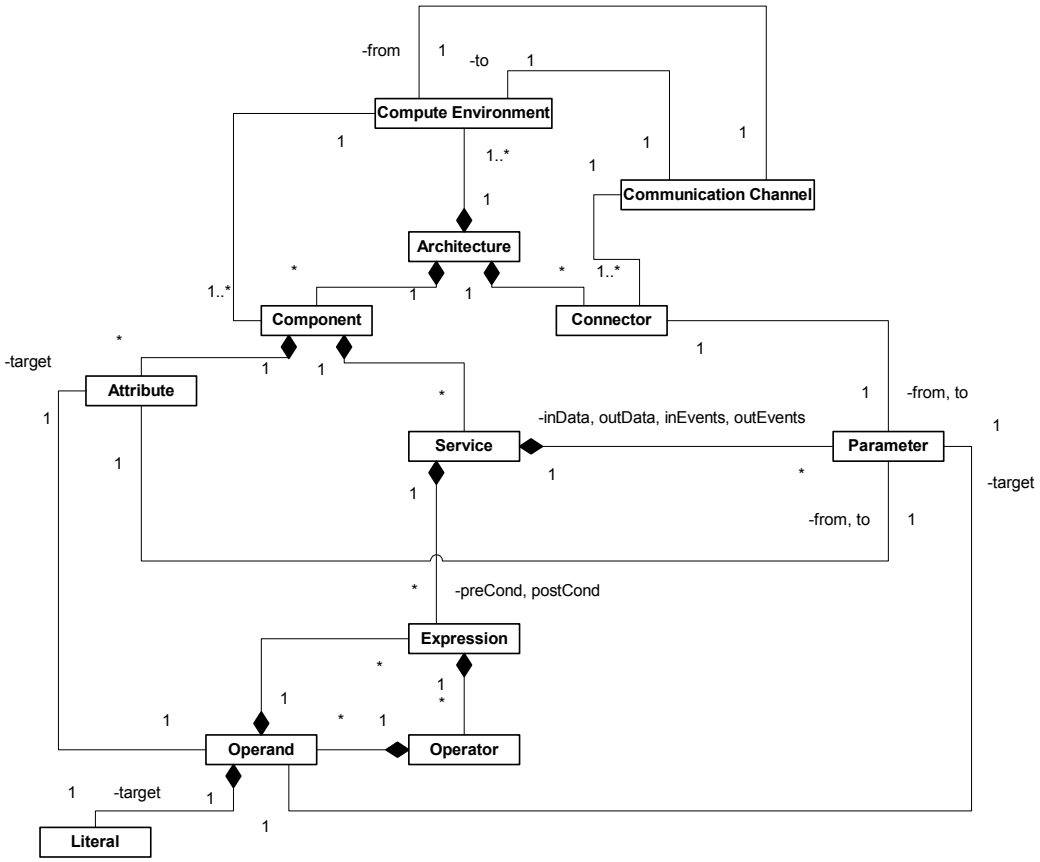


Figure 15 - Arcade Architecture Metamodel

For all subsequent formal definitions, the following notational conventions will be used. Structural elements in the Arcade metamodel will be defined in the form `x:Type` or `X:{Type}` which declares an element named `x` of type *`Type`* or a collection of elements named `X` of type *`Type`*, respectively. Types of elements include entities shown in Figure 15, including components, services, and attributes. Elements declared of type *`Type`* may contain one or more named fields, where each field is declared as a particular element type or a collection of elements of that type. An individual field under a declared element can be referenced through the dot operator ("`.`"). For instance, field $x_1$ of declared element `x` can be referenced using the expression $x.x_1$. If an element belongs to a specific collection defined elsewhere in the representation, the relationship is noted accordingly (e.g., $x \in X$).

#### *3.2.2.2 ARCH Representation*

The Arcade Architecture (`ARCH`) can be described by the 5-tuple, `(n,C,Q,E,X)`, shown in Figure 16. An `ARCH` is uniquely identified by a name, `arch.n`. Each `ARCH` has associated collections of components, `arch.C`, and collections of connectors `arch.Q`. A set of compute environments, `arch.E` and a set of communication channels, `arch.X` are also defined. The representation for components, connectors, compute environments, and communication channels will be defined in the following sections.

| arch:*ARCH* ≝ (n,C,Q,E,X) |
|---|
| n:*String*<br>name of the architecture.<br><br>C:{*Component*}<br>set of all components in arch.<br><br>Q:{*Connector*}<br>set of all connectors in arch<br><br>E:{*Compute Environment*}<br>set of all compute environments in arch<br><br>X:{*Communication Channel*}<br>set of all communication channels in arch |

Figure 16 - ARCH representation

#### *3.2.2.3 Component Representation*

A component is represented as the 3-tuple: (n,A,S), shown in Figure 17. Each component is identified by a unique name in the ARCH (c.n), and contains collections of attributes (c.A) and services (c.S).

| c:*COMPONENT* ≝ (n,A,S) ∈ arch.C |
|---|
| n:*String*<br>name of the component. The name must be unique among all components in the respective ARCH.<br><br>A:{*Attribute*}<br>set of attributes owned by component c.<br><br>S:{*Service*}<br>set of services provided by component c. |

Figure 17 - Component Representation

#### *3.2.2.4 Attribute Representation*

Each attribute owned by a component, `a ∈ c.A`, can be described by the 3-tuple, `(n,t,A)`, in Figure 18. The attribute type, `a.t`, characterizes an attribute as either an event or a data attribute; data types include one of the following: (i) a primitive data type (e.g., `String`, `Integer`, or `Float`), (ii) "Composite," or (iii) "Unspecified." A "Composite" data attribute is "composed of" other data attributes. The constituent attributes for a "Composite" attribute are referenced through the field `a.A`. Constituents must belong to the same component as the composite, and attributes may not be recursively composed.

| $a:Attribute \stackrel{def}{=} (n,t,A_c) \in c.A$ |
|---|
| `n:`*`String`*<br>name of the attribute.<br><br>`t` type of the attribute. If the attribute is composed of other attributes (see child attributes below), it carries a type of "Composite."<br>`a.t ∈ {Event,String,Integer,`<br>`Float,Composite,Unspecified}`<br><br>`A:{`*`Attribute`*`}`<br>set of constituent attributes (if respective attribute is "Composite"). |

Figure 18 - Attribute Representation

#### *3.2.2.5 Service Representation*

Each service $s \in c.S$ is described by the 8-tuple, $(n, \delta, D_i, D_o, E_i, E_o, C_{Pre}, C_{Post})$, in Figure 19. Average duration, $s.\delta$, specifies the typical time required for service execution in unit time (e.g., relative to other services in ARCH). Service input and output data/events are described by fields, $s.D_i$, $s.D_o$, $s.E_i$, and $s.E_o$. The service pre-conditions, $s.C_{Pre}$, are first order logic expressions that specify the conditions necessary to initiate execution of the service, including the required input data and events. Similarly, the service post-conditions, $s.C_{Post}$, are first order logic expressions that describe the state of the architecture after service execution has completed, including the output data produced and the events generated.

$s$:*Service* $\triangleq$ $(n, \delta, D_i, D_o, E_i, E_o, C_{Pre}, C_{Post}) \in c.S$

$n$:*String*
name of the service.

$\delta$: *Integer*
average service duration.

$D_i$:{*Parameter*}, $D_o$:{*Parameter*},
$E_i$:{*Parameter*}, $E_o$:{*Parameter*}
input data, output data, input events, and output events

$C_{Pre}$:*SvcPreCond*
first order logic expression describing pre-conditions on the execution of this service.

$C_{Post}$:*SvcPostCond*
first order logic expression describing post-conditions on the execution of this service.

Figure 19 - Service Representation

#### *3.2.2.6 Parameter Representation*

Figure 20 describes the parameter metamodel element used to specify service input data, output data, input events, and output events. Each parameter $p \in s.D_i \mid s.D_o \mid s.E_i \mid s.E_o$ is described by the 2-tuple, $(n,k)$. Each parameter is identified by a name ($s.d_i.n$, $s.d_o.n$, $s.e_i.n$, and $s.e_o.n$) that uniquely identifies it among other inputs and outputs within a service. The fields $s.d_i.a$, $s.d_o.a$, $s.e_i.a$, and $s.e_o.a$ reference the respective input or output data or event for a service, where these data and events are defined as attributes.

| $p$: *Parameter* $\stackrel{def}{=} (n,k) \in s.D_i \mid s.D_o \mid s.E_i \mid s.E_o$ |
|---|
| $n$: *String*<br>name of the parameter (referenced in pre-conditions).<br><br>$k$<br>type of parameter.<br>$k \in \{inData, outData, inEvent, outEvent\}$ |

Figure 20 - Parameter Representation

#### *3.2.2.7 Expression Representation*

Expressions are composed of predicates and are used to specify service pre- and post-conditions. The predicates that can be used in expressions are shown in Figure 21.

$C_{Pre}$:*SvcPreCond*,$C_{Post}$:*SvcPostCond*

Service pre-conditions and post-conditions are expressed as first order logic expressions utilizing one or more of the following predicates:

- Data value comparison (i.e., particular data value is equal to, less than, greater than some other value), where the value of a data attribute is referenced using `DATA_VALUE( <Component name>,<attribute name> )`. For example, if a condition required attribute "A1" in Component "C1" to be greater than 5, the pre-/post- condition would be expressed as `DATA_VALUE("C1","A1") > 5`.

- Availability of service input data, expressed as: `INPUT_DATA_AVAILABLE( <unique service input data name> )`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.

- Presence of triggering service input event, expressed as: `INPUT_EVENT_GENERATED(<unique service input event name>)`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.

- Production of service output data, expressed as: `OUTPUT_DATA_AVAILABLE(<unique service output data name>)`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.

- Generation of service output event, expressed as: `OUTPUT_EVENT_GENERATED(<unique service output event name>)`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.

Figure 21 - Expression Representation

#### *3.2.2.8 Connector Representation*

Each connector owned by the architecture, $c \in arch$, can be described by the 5-tuple, $(n, P_f, P_t, A_f, A_t)$, in Figure 22. The source parameter for a connector may be either NULL (e.g., the source is external to arch), or may be associated with a single output parameter of a service. The destination parameter for a connector may be either NULL (e.g., the destination is external to arch), or may be associated with a single input parameter of a service. The source attribute for a connector may be either NULL (e.g., the source is external to the arch), or may be associated with a single attribute of a component. The destination attribute for a connector may be either NULL (e.g., the destination is external to arch), or may be associated with a single attribute of a component.

$q$:*Connector* $\stackrel{def}{=} (n, P_f, P_t, A_f, A_t) \in arch.Q$

$n$:*String*
name of the connector.

$P_f$:*Parameter*
Source parameter for this connector.

$P_t$:*Parameter*
Destination parameter for this connector.

$A_f$:*Attribute*
Source attribute for this connector.

$A_t$:*Attribute*
Destination attribute for this connector.

Figure 22 - Connector Representation

#### *3.2.2.9 Compute Environment Representation*

A compute environment `e` is represented as the 4-tuple, `(n,C,X,s)`, shown in Figure 23. A compute environment represents a hardware environment in which components execute. Each compute environment is identified by a unique name in its `ARCH` (`e.n`). Compute environments contain collections of components (`e.C`) and are associated with one or more communication channels (`e.X`). Each compute environment also defines its execution speed (`e.s`, relative to other compute environments in its `ARCH`).

| `e:`*`Compute Environment`* ≝ `(n,C,X,s) ∈ arch.E` |
|---|
| `n:`*`String`*<br>name of the compute environment. The name must be unique among all compute environments in its `ARCH`.<br><br>`C:{`*`Component`*`}`<br>set of components associated with compute environment `e`.<br><br>`X:{`*`Channel`*`}`<br>set of communication channels associated with compute environment `e`.<br><br>`s:`*`Integer`*<br>relative execution speed of compute environment `e`. |

Figure 23 - Compute Environment Representation

#### *3.2.2.10 Communication Channel Representation*

A communication channel `x` is represented as the 4-tuple, `(n,p,q,s)`, shown in Figure 24. A communication channel represents a bi-directional hardware link between two compute environments. Each communication channel is identified by a unique name in its `ARCH` (`x.n`), and contains an association with two compute environments (`x.p`) and (`x.q`). Each communication channel also defines its latency (`x.s`, relative to other communication channels in `ARCH`).

| x:*Communication Channel* $\stackrel{def}{=}$ (n,p,q,s) $\in$ arch.X |
|---|
| n:*String*<br>name of the communication channel. The name must be unique among all communication channels in the respective `ARCH`.<br><br>p,q:*Compute Environment*<br>pair of compute environments associated with `x`.<br><br>S:integer<br>relative latency of communication channel x. |

Figure 24 - Component Representation

The following sections describe the approach for translating each of the SEPA3D Architecture models to Arcade models.

#### *3.2.2.11 Translating the Domain Reference Architecture to the Arcade Metamodel*

Elements of a Domain Reference Architecture (DRA) are translated to the Arcade metamodel by mapping DRA elements to complementary elements of the Arcade metamodel. Arcade components are constructed by mapping DRACs in the DRA along with associated DRA services and attributes. DRA services include pre-/post-conditions and input/output data and event definitions; these are mapped to their respective Arcade services. Arcade connectors are created for each service and attribute dependency in the DRA, creating connections between components based on required exchanges of events and data. A detailed algorithm for this process is included in APPENDIX B.

#### *3.2.2.12 Translating the Application Architecture to the Arcade Metamodel*

Translating an Application Architecture (AA) to the Arcade metamodel is very similar to translating a DRA to the Arcade metamodel. The primary distinction is that the AA is defined in terms of Technology Solutions (TS) instances rather than DRACs. These TS instances have mappings to elements of the DRA against which they are registered (Section 2.2.2). The translation process is therefore one of translating TSs in the AA to Arcade Components, and then following the registration mappings between the AA and DRA to populate the remainder of the Arcade metamodel using the algorithm defined in Section 3.2.2.11.

#### *3.2.2.13 Translating the Implementation Architecture to the Arcade Metamodel*

The Implementation Architecture (IA) contains information regarding the additional TSs available at an installation site to satisfy site installation requirements as well as infrastructure requirements in the AA (Section 2.2.3). This information is used to associate TSs from the AA with compute environments. Once the TSs have been allocated to compute environments, the dependencies between services are examined to determine which communication channels must exist between compute environments (e.g., services which must exchange data and/or events must have a communication channel between their respective compute environments). The remainder of the IA translation process follows the AA translation process described in Section 3.2.2.12.

Once a SEPA 3D Architecture model has been translated to an Arcade model, a number of dynamic property evaluations become available using the Arcade evaluation process described in Section 3.1.3. The following sections discuss each of the dynamic properties that Arcade can evaluate, and the techniques used to evaluate the properties.

### 3.2.3 Correctness Evaluation

Arcade provides an automated combination of simulation, visualization, and model checking for evaluation of *safety*, *liveness*, and *completeness* properties [11, 17]. Utilizing information contained in the Arcade model, some forms of correctness properties can be expressed to evaluation tools on behalf of

stakeholders without requiring specialized knowledge of the underlying correctness evaluation tools. This automated property expression technique will be described in detail in the following sections.

Arcade employs an off-the-shelf simulation engine and model checking tool called SPIN [61]. SPIN can perform both model checking and simulation using a single input model. Therefore only one translation step is required for all types of correctness properties that Arcade can evaluate. Arcade presents correctness evaluation results to stakeholders using a number of different graphical and textual notations. These presentation techniques are described in the following sections along with their associated correctness property.

SPIN has been applied to both research and industrial problems. Research applications include real-time verification, reactive systems modeling, and extending process algebra tools to support correctness evaluations. Industrial applications include checking correctness of distributed algorithms, communications network design problems, and various hardware and software protocols. Domains to which SPIN have been applied include distributed transaction systems, microkernel design, security protocols, multiprocessor designs, railway signaling protocols and hardware, distributed component frameworks, and many telecommunications protocols [27, 50, 92].

While most of the interaction between Arcade and SPIN is automated, Arcade allows stakeholders to specify one model checking parameter to be passed to SPIN. This parameter is `MAX_ERRORS`, indicating the maximum number of

safety or liveness errors that SPIN should allow before terminating a model checking session.

The following sections discuss how Arcade models are translated to Promela, how correctness properties are defined and evaluated, and how evaluation results are presented to stakeholders.

#### *3.2.3.1 Translation from Arcade Metamodel to Promela*

The SPIN model checker accepts models expressed in a language called Promela. In general, the tasks that must be accomplished when translating a software architecture to Promela are:

1) Identifying which SPIN processes to create and how architectural elements will map to these processes (including what data and events are owned and managed by each process),
2) Specifying how each SPIN process behaves, and
3) Specifying what data and events are exchanged between SPIN processes and how they are exchanged (e.g., rendezvous, message passing, or shared variables).

Each of these tasks is automated with respect to an architecture represented in the Arcade metamodel. The mapping from the Arcade metamodel to Promela is depicted in Figure 25 and is described in more detail in the

following paragraphs. A detailed sketch of the Arcade to Promela translation algorithm is presented in APPENDIX C.

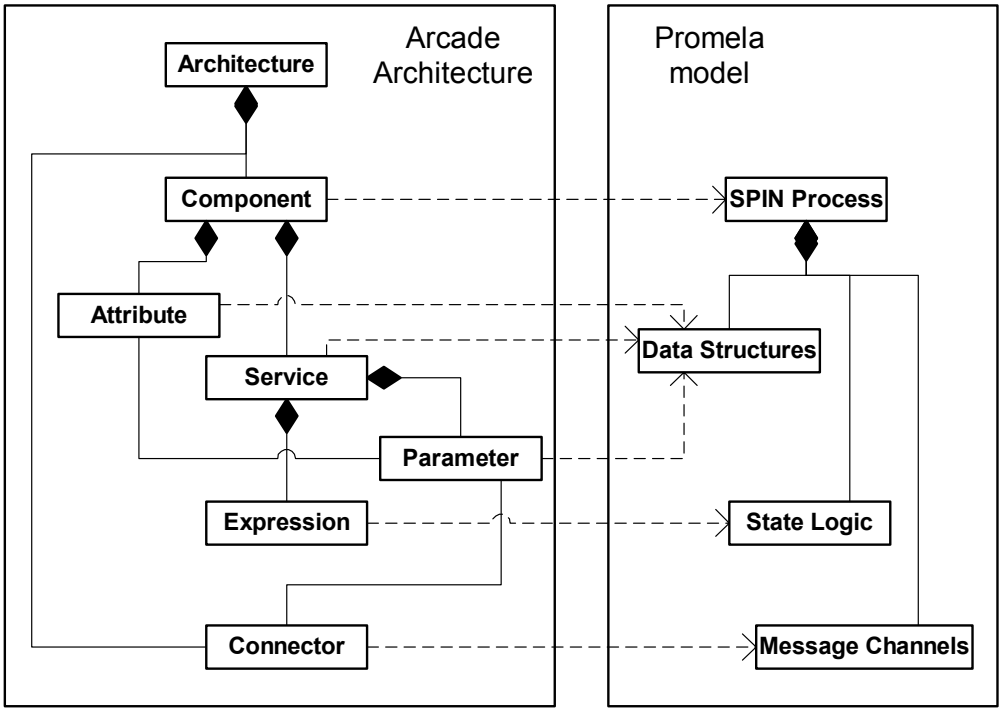


Figure 25 - Translation From Arcade Metamodel to Promela

Each component in the Arcade model has a set of services and a set of attributes. Each service has a set of input and output data and events, and a set of pre-conditions and post-conditions. Arcade uses this information to generate:

1) Promela code that creates a SPIN process for each component in the architecture (i.e., each component becomes one SPIN

process, such that component "COMPONENT1" is translated to Promela code to implement the SPIN process "COMPONENT1_Model"),

2) A set of Promela data structures to associate component attributes with the correct SPIN process (i.e., data and events owned by a given component are translated into data structures associated with the respective SPIN process), and
3) A set of Promela data structures to associate component services and service input and output data and events with the correct SPIN process (i.e., descriptions of service parameters are translated into data structures associated with the respective SPIN process).

Arcade translates the pre- and post-conditions of services into state logic contained within each SPIN process corresponding to the component that owns the service. Predicates in the pre- and post-conditions are modeled as guards that either block or enable other statements for execution. This is done using the Promela `do` construct within the Promela processes that represent components. The `do` construct functions as an endless loop, executing one of its non-blocked statements each time through the loop (or blocking until at least one statement becomes executable). Execution continues until no statements in the Promela model are executable.

The *do* construct allows a component to be "IDLE" (e.g., blocked in the *do* construct) until one or more of the pre-conditions for its services becomes enabled (executable). If more than one pre-condition is executable, the *do* construct allows SPIN (or an external controller such as Arcade) to make a (possibly non-deterministic) choice about which service to execute. A Promela psuedo-code example from the eDesign DRA is:

```
process Product_Owner_Model {
  do
  // code for "Change Product State" process
  :: ("Change Product State" pre-condition predicate)
       -->
       ("Change Product State" service-execution code)
       ("Change Product State" post-condition predicate)
  // similar code for other services
  ...
  od
}
```

When a pre-condition becomes executable, the component transitions to an "EXECUTING" state for the corresponding service. Eventually, the post-condition for the service becomes executable (assuming that the model is correct and the post-condition can trigger), and the component returns to an "IDLE" state.

Promela processes run concurrently and may exchange data and events over channels. Therefore, a Promela message channel is created for each connector in the Arcade architecture, and Promela code is also generated to connect the interfaces between services in the components by making the appropriate SPIN channel assignments. To allow for complex predicates, the Promela method of checking channels for messages without consuming them is used in implementing the pre-condition logic; therefore the first action performed after the predicate is satisfied is to consume the events and data; the general syntax for this is:

```
// code for a service:
// pre-condition predicate check
// followed by data/event consumption
:: (chan?[message]) ->
     chan?message;
     (service-execution code)
     (post-condition predicate)
```

Events and data referenced in disjunctive pre-conditions are consumed by placing them in a non-deterministic *if* construct following the pre-condition check. This prevents the process from becoming blocked in an attempt to consume data/events that are not present (since the pre-condition is disjunctive). Disjunctive post-conditions are also generated using an *if* statement.

Two SPIN processes are created to close the model (allowing the external world to be modeled within SPIN):

- Scenarios - generates initial events/data for threads of execution; and
- External - receives and generates all events/data that are specified as "external" in the architecture.

Input events or data that are specified as "external" are received from entities outside the software architecture model (e.g., humans, machines, or other software systems). If a service has only "external" data and/or event references in its pre-condition, these data and events will automatically be generated by the *Scenarios* process upon simulation startup. If a pre-condition has references to both internal and external input events/data, the *External* process will monitor for the existence of required internal data/events and upon detecting them it will generate the external events/data identified in the pre-condition. The predicates for generating these events/data include a check to determine if the external events/data are already in a channel to prevent false generation of duplicate events/data before the consumer has a chance to consume the data/events.

The following sections discuss in more detail each of the correctness properties evaluated by Arcade, how they are evaluated, and how evaluation results are presented to stakeholders.

### 3.2.4 Safety Property Evaluation

This section describes safety property evaluation within Arcade. The discussion includes the safety property definition, how Arcade defines the safety property in Promela, and how results are presented to stakeholders.

#### *3.2.4.1 Safety Property Definition*

In the context of Arcade evaluation, safety is informally defined as "the system never terminates in an undesirable end state" [72, 76]. Undesirable end states include unterminated service executions, and the presence of unconsumed events or data. Therefore, Arcade verifies that the following conditions hold: (1) if the pre-condition of a service has been satisfied, the service will eventually execute, (2) if a service executes, eventually its post-condition will be satisfied, and (3) all events and data produced as outputs of a service are eventually consumed. This property is expressed formally as follows:

$$
\begin{aligned}
\Box\ &( \forall\, s{:}\ \mathit{Service} \in a{:}\ \mathit{ARCH} \mid \\
&\quad preCondition(s) \rightarrow \\
&\quad \Diamond\, executed(s) \rightarrow \\
&\quad \Diamond\, postCondition(s)\ \wedge \\
&( \forall\, e{:}\ \mathit{SvcOutEvt} \in s.E_{\circ} \mid \\
&\quad produced(e) \rightarrow \Diamond\, consumed(e)\ )\ \wedge \\
&( \forall\, d{:}\ \mathit{SvcOutData} \in s.D_{\circ} \mid \\
&\quad produced(d) \rightarrow \Diamond\, consumed(d)\ )\ )
\end{aligned}
$$

Valid end states are signified in Promela by the presence of a label whose identifier begins with the string “end.” Arcade labels all *do* constructs in

component processes as valid end states (recall that a component process blocked in its *do* construct is "IDLE"). Therefore, this labeling signifies that a component that is "IDLE" is in a valid end state. If the component process terminates somewhere within the body of the *do* construct, this signifies an invalid end state (e.g., the component terminated in a pre-condition satisfaction state, in a service execution state, or in a post-condition satisfaction state.) An example *do* construct labeled for safety checking is shown below.

```
process Product_Owner_Model {
 // blocked in "do" is a valid end-state
 end_Product_Owner_Model:
  do
  // code for "Change Product State" service
  :: ("Change Product State" pre-condition predicate)
       -->
       ("Change Product State" service-execution code)
       ("Change Product State" post-condition predicate)
   // code for other services
   . . .
  od
 }
```

#### *3.2.4.2 Execution: Model Checking the Safety Property*

SPIN supports model checking by generating a model-specific verifier program in the C language. To perform safety evaluation, Arcade instructs SPIN to use the Arcade-generated Promela model to create a verifier for the architecture, and then Arcade compiles and invokes the verifier to check the safety property. The verifier notifies Arcade of property violations that have been detected. For safety evaluations, the verifier creates counter-examples of system executions that exhibit safety violations.

Counter-examples of safety property violations can be examined using the off-the-shelf SPIN tool set. This is done using the XSpin user interface to produce a visualization of the property violation in the form of a Message Sequence Chart (MSC) [61, 65]. However, these MSCs contain detailed Promela-specific information that is difficult to interpret without (1) specialized knowledge of SPIN and Promela, and (2) knowledge of the structure and semantics of the Promela code representing the architecture being verified [83]. Arcade addresses this limitation by replacing Promela-specific information with information that is expressed in architectural terms. The following section discusses the Arcade approach for presenting safety errors to stakeholders in a way that does not require model checking expertise.

#### *3.2.4.3 Presentation: Architecture Trace Diagrams*

To reduce the complexity of model checker output, Arcade presents a SPIN counter-example using a graphical representation called an Architecture

Trace Diagram (ATD). ATDs are based on the ITU Message Sequence Chart standard [65]. MSC diagrams are widely used in industry, and are useful for depicting interactions between communicating processes. The ATD extensions are intended to make Arcade results easily interpretable for stakeholders who are not experts on model checking or Promela, rather who are experts on the requirements captured in an architecture. To accomplish this, the ATD definition implements extensions to the MSC and eliminates un-needed features. The areas of extension are related to message passing, and actions. Excluded MSC features include conditions, instance ends, timers, co-regions, process creations, and process stops.

The standard MSC message notation is an arrow drawn between two process instance axes (lifelines), annotated with the message being passed. This annotation method is impractical to represent data or event exchanges for an Arcade architecture, since the Arcade metamodel allows a sender to identify the data or event being sent using a different name than the receiver uses (this was necessary in order to support decoupling of sent and received data and events in the DRA metamodel). Given this requirement, ATDs implement message exchange visualization differently than the MSC standard, using (1) a state on the sender's lifeline identifying the data or event being sent using the sender's terminology, (2) a state on the receiver's lifeline identifying the data or event being received in the receiver's terminology, and (3) an un-annotated arrow connecting the two states.

The MSC "actions" (in this case, component states) to be displayed on the ATD come from a fixed set of state classes (e.g., pre-condition satisfaction, service execution, etc.). For clarity, it is desirable to depict a component state in a way that is intuitive, unambiguous, and quickly distinguishable from within this set of component states. Given this requirement, ATDs define a set of diagram elements for specific component states rather than simply using an "action" box to depict these states as a standard MSC would. An example ATD is shown in Figure 26. The legend in this example describes each specific ATD state and how it is depicted in an ATD. The full ATD definition is detailed below.

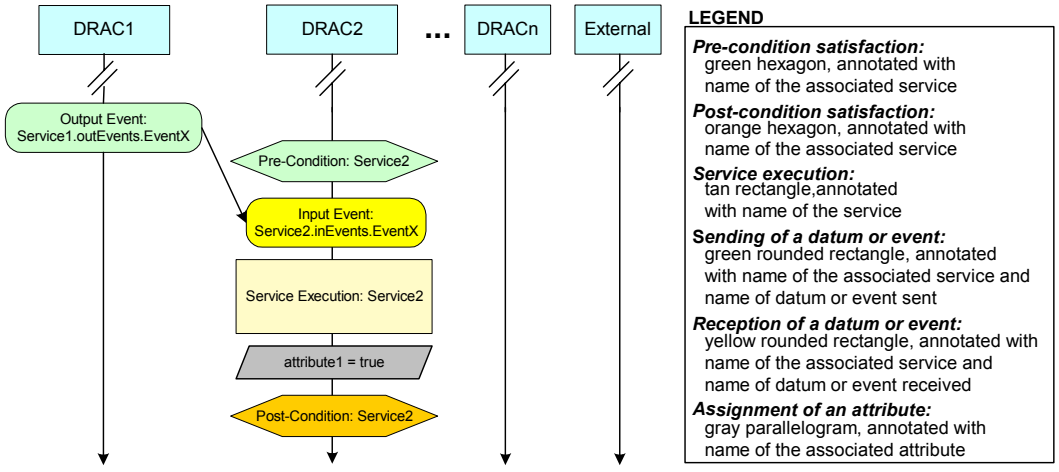


Figure 26 - Example ATD Diagram

An Architecture Trace Diagram (ATD) has two dimensions: (1) the vertical dimension represents time (top to bottom), and (2) the horizontal

dimension represents components involved in the trace. There is no significance to the horizontal ordering of these processes. An ATD represents each component involved in the trace using a rectangle indicating the name of the component at the top of the diagram (e.g., "COMPONENT1," "COMPONENT2," "External"). Each component in an ATD has an associated lifeline that proceeds in the vertical dimension from top to bottom. Trace states are arranged in time sequence along the lifeline of the component with which they are associated. The relative ordering of states along lifelines is significant, but the distance between states does not indicate duration. Corresponding send and receive states in the ATD have a directed arc drawn from the sending state to the receiving state. These arcs may cross if the order for sending a set of events is different from the order in which they are received.

Any property violation(s) depicted in the ATD are represented by the normal shape associated with the state class that has caused the property violation, but with the figure shaded red instead of the normal color associated with the event type. Property violations are also annotated with a message indicating the type of error and what caused the error.

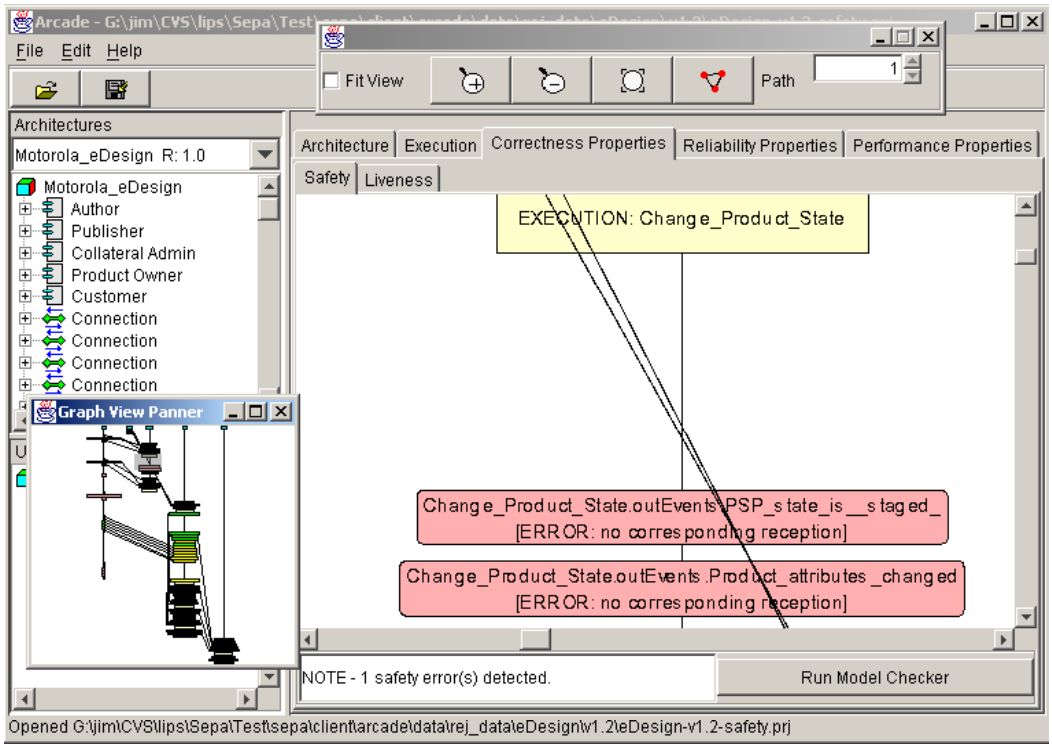


Figure 27 - Arcade ATD Display

Figure 27 shows an Arcade ATD zoomed in to view safety errors discovered in the eDesign DRA ("ERROR: no corresponding reception"). The accompanying panner window shows a thumbnail image of the ATD. Safety errors were caused by two unconsumed events ("PSP state is staged", and "Product attributes changed") that have been output by the "Change Product State" service of the "Product Owner" DRAC. These events were output as specified by the post-condition of the "Change Product State" service, but were not subsequently consumed by any pre-condition. When this type of safety error occurs, the architect and stakeholders must determine whether the errors occurred

as a result of (1) an invalid post-condition (e.g., the “PSP state is changed” and “Product attributes changed” events should not have been generated in the post-condition for the “Change Product State” service), or (2) as a result of an invalid pre-condition associated with other service(s) of the architecture (e.g., some other service requires the “PSP state is staged” and “Product attributes changed” events, but this requirement was not specified).

### 3.2.5 Liveness Property Evaluation

This section describes liveness property evaluation within Arcade. The discussion includes the liveness property definition, how Arcade defines the liveness property in Promela, and how results are presented to stakeholders.

#### *3.2.5.1 Liveness Property Definition*

The liveness property that Arcade verifies is informally defined as "the system eventually enters all desirable states" [72, 76]. An architecture has no liveness errors for this property when the following conditions hold: (1) no unreachable services exist, and (2) all required paths between services are traversable. Unreachable services occur when an entire pre-condition is never satisfiable. Untraversable paths occur when a disjunct sub-expression of a pre-condition is never satisfiable.

Formally, the liveness property evaluated by Arcade is defined as follows:

$$\Box\,(\ \forall\, s{:}\ \mathit{Service} \in a{:}\ \mathit{ARCH}\ \mid \\ \forall\, x{:}\ \mathit{SvcPreCond} \in \mathrm{disjuncts}(s.C_{Pre})\ \mid \\ \Diamond\, x\ )$$

Checking of *liveness* properties is enabled by instructing SPIN to search for unreachable states in the Promela model of the architecture. This search requires no additional information (beyond that required for safety evaluations) to be supplied to SPIN via Promela or LTL expressions.

The following section describes how Arcade uses SPIN to verify the liveness property.

#### *3.2.5.2 Execution: Model Checking the Liveness Property*

Arcade checks for the liveness property defined in Section 3.2.5.1 by instructing SPIN to generate a liveness property verifier for an architecture. Once the verifier has been created, Arcade invokes it to check for unreachable service execution states. These states indicate pre-conditions in the architecture that cannot be satisfied. The SPIN verifier notifies Arcade of states that cannot be reached in terms of the specific line of Promela code that cannot be reached. As with safety property violation counter-examples, the details of the Promela code are too low-level for stakeholders to interpret without specialized knowledge of Promela, and how the architecture was translated into SPIN input. Therefore,

Arcade uses its knowledge of the architecture contents and the Promela translation process to interpret these low-level Promela states and present information back to the stakeholders in terms of familiar architectural elements.

#### *3.2.5.3 Presentation: Unsatisfiable Pre-conditions*

Arcade correlates lines of Promela code reported as unreachable by the SPIN verifier with the associated pre-condition expressions in the architecture model. These pre-conditions are predicates that must be made satisfiable in order for the un-reachable line of Promela code to become reachable. Once an unsatisfiable pre-condition has been identified within the original architecture model, Arcade presents a formatted text version of the pre-condition expression to the Arcade user (Figure 28).

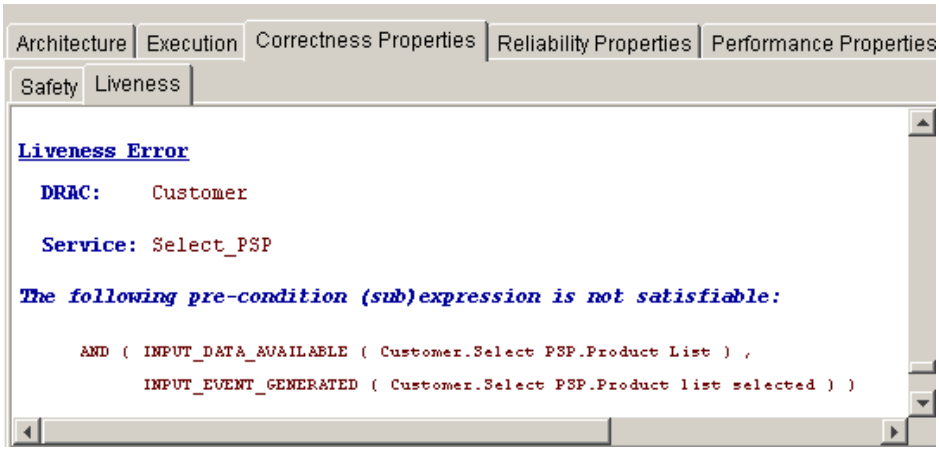


Figure 28 - Arcade Presentation of Liveness Errors

The display in Figure 28 shows Arcade presenting an unsatisfiable pre-condition in the eDesign DRA. This liveness error occurred for the “Select PSP” service in the “Customer” DRAC. This situation can occur when one or both of the input data/event are marked as being required from another service in the architecture, yet were never produced by any service in the architecture.

### 3.2.6 Completeness Property Evaluation

This section describes completeness property evaluation within Arcade. The discussion includes the completeness property definition, how Arcade defines the completeness property in Promela, and how results are presented to stakeholders.

#### *3.2.6.1 Completeness Property Definition*

The completeness property that Arcade can help evaluate is informally defined as "an architecture reflects the functionality required by stakeholders." In other words, threads of execution specified within an architecture must reflect sequences of service executions that domain experts require (i.e., domain usage profiles), with no missing or extraneous service executions. Unfortunately, model checking tools such as SPIN cannot check that a software architecture provides correct semantics (completeness) without requiring additional properties to be defined in a language such as LTL [61]. This requirement increases the need for expertise when performing completeness evaluations with a model checker.

Arcade bypasses this expertise issue by taking advantage of the following observations. Completeness errors are associated with unexpected behavior of the system, where behaviors are described by respective usage profiles. These errors are typically manifested in sequences of service executions (usage profiles) that: (1) are missing expected service executions, (2) contain unexpected service executions, (3) contain unexpected paths, or (4) are missing paths. Therefore, to assist the architect in verifying dynamic completeness, Arcade employs the SPIN guided simulation feature [61] to generate a visualization of threads of execution, referred to in this research as an Execution Space (Section 3.2.6.3). Rather than specifying LTL expressions, domain experts can inspect an Execution Space to detect completeness errors. While this evaluation approach can be very complex for a large architecture, experience with the eDesign case study indicates that partial model evaluations supported by Arcade can make this task manageable (to be discussed more in Chapter CHAPTER 4).

#### *3.2.6.2 Execution: Guided Simulation*

Arcade performs completeness evaluation by iterating over the following steps: (1) simulating architecture execution and merging all states occurring during simulation into a low-level state space (each simulation run creates a unique path in the low-level state space), and (2) performing a partial order reduction over the low-level state space with respect to service execution states. Each iteration explores a different thread of execution (path) allowed by the Promela model.

Arcade controls the SPIN simulation engine as a child process to generate the low-level state space. Stakeholders can choose to do this using either an exhaustive path selection or random path selection approach. Using either path selection mode, Arcade successively explores execution threads, capturing all states that occur, and keeping track of which paths have been explored.

A path in the low-level state space is a total order representing one execution of the architecture. A branch in the low-level state space occurs when alternate interleavings of simulation events are possible (for example different interleaving of sending or reception of events, or in the presence of disjunctive pre-/post-conditions).

Following exploration of a path in the low-level state space, Arcade performs partial order reduction of the low-level state space into the Execution Space (ES) by merging paths that share identical partial orders of service execution states. Therefore, each path in the ES represents an equivalence class for a set of paths in the low-level state space. (e.g., low-level state space paths in this equivalence class share a common partial order of service execution states). The entire ES represents the set of all equivalence classes for all paths in the low-level state space (assuming all paths in the low-level state space have been explored).

Partial order reduction techniques have been used in prior software architecture research, most notably in Rapide [85]. Arcade extends this concept by constructing the ES incrementally. This approach allows the architect to visualize the execution of the system as early as possible rather than requiring the

architect to wait until the entire low-level state space has been explored. For each path explored in the low-level state space, incremental merging into the Execution Space is accomplished using the following algorithm:

```
Algorithm: partialOrderMerge(ES: Execution Space,
                             S: {State} )

// S is the totally ordered sequence of states from
// the most recently explored path in the
// low-level state space: {INIT, s0, …, sn, TERM}

m: State  = S[0]
repeat while S ≠ {}
   P: {State} = max length path prefix of ES starting
                from m that matches a prefix of S
   m = last service execution state in P
   S = S - P
   Q: {State} = prefix of S containing only service
                execution states not in ES
   S = S - Q
   merge(ES,m,Q)   // merge Q into ES starting at m
   m = service execution state in ES matching first
       service execution state in S
```

The following section describes how the ES is presented to Arcade stakeholders during completeness evaluation.

#### *3.2.6.3 Presentation: the Execution Space*

The Execution Space (ES) provides a high-level view of the low-level state space resulting from SPIN guided simulations. The low-level state space is a directed graph of states and transitions that are possible during execution of the architecture. These states include exchange of events and data, changes in data values, satisfaction of pre-conditions and post-conditions, and service executions. This low-level state space can become very large and difficult to visualize.

An ES is a directed graph representing threads of service executions allowed by an architecture. A vertex in the graph represents a service execution state, and an edge represents a path from one service execution state to another. Two additional vertices in the ES are a super-initial state (INIT), and a super-final state (TERM). These vertices provide common initiation and termination states for paths in the ES.

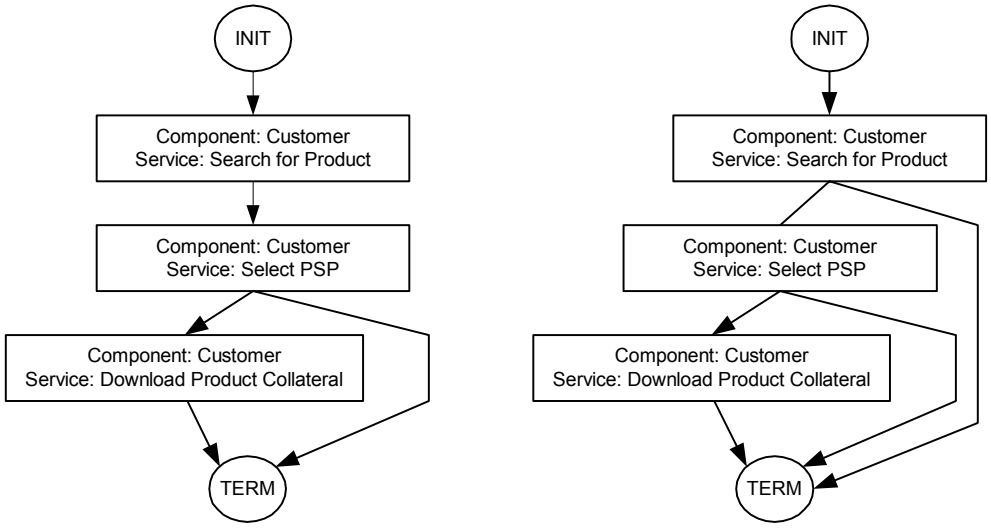


Figure 29 - (a) Execution Space With Error; (b) Corrected

Figure 29(a) illustrates an ES for a DRA version containing a partial model of the eDesign DRA (the "Access Product Information" usage profile, Chapter CHAPTER 4). An eDesign stakeholder identified a completeness error in this ES, noting that there was no path from the "Search for Product" service to the TERM node. This path should exist to support the domain requirement to allow a customer to cancel their product search. A corrected eDesign DRA version produced the Execution Space in Figure 29(b).

The following section presents the approach and implementation of performance evaluation in Arcade.

### 3.2.7 Performance Evaluation

Arcade uses the Simpack discrete event simulation toolkit as the basis for its performance evaluations [18, 47]. The simulation approach defines usage profiles as units of work that enter the system and must be executed. Performance metrics are captured from Simpack while usage profiles are initiated and executed. Although the main unit of work is the usage profile, statistics are gathered for usage profiles, services, components, and compute environments. The overall approach is to perform a number of repeated simulation runs with the system becoming increasingly loaded over the course of simulation runs. The resulting metrics can be used to identify performance critical elements of the architecture.

To support this approach, Arcade allows stakeholders to specify several simulation parameters when performing simulation. These parameters include: (1) `IA_TIME`: the interarrival time for usage profiles (used to control system loading), (2) `TRIALS`: number of simulation runs to perform for gathering summary statistics, and (3) `SIM_TIME`: maximum simulation time. When all simulation runs have completed, Arcade reports summary statistics.

The following section defines the performance metrics that Arcade can evaluate. This is followed by discussions of how Arcade uses Simpack, and how results are presented to stakeholders.

### 3.2.8 Usage Profile Performance Properties

Arcade can evaluate include the following usage profile performance properties:

- *Usage Profile Latency* - mean execution time of a usage profile; expressed as follows:

$$L_{UP} = \frac{\sum_{i=1}^{I_{UP}} T_{UP_i}}{I_{UP}}$$

where:

$UP$ is a usage profile;

$T_{UPi}$ is execution time of $UP$ invocation $i$; and

$I_{UP}$ is number of $UP$ invocations

- *Usage Profile Throughput* - executions/unit time of a usage profile; expressed as follows:

$$TP_{UP} = \frac{I_{UP}}{T_{SIM}}$$

where:

$UP$ is a usage profile;

$I_{UP}$ is number of $UP$ invocations; and

$T_{SIM}$ is total simulation time

### 3.2.9 Component Performance Properties

Components are defined as either DRACs or TSs (for a DRA component ≡ DRAC; for an AA or IA component ≡ TS). Arcade can evaluate the following component performance properties:

- *Component Utilization* - amount of time spent performing service contained in component; expressed as follows:

$$U_C = \frac{T_{SVC}}{T_{SIM}} \mid SVC \in C$$

where:

$C$ is a component;

$SVC$ is a service;

$T_{SVC}$ is total time spent executing $SVC$; and

$T_{SIM}$ is total simulation time

- *Component Throughput* - executions/unit time of a component; expressed as follows:

$$TP_C = \frac{I_C}{T_{SIM}}$$

where:

$C$ is a component;

$I_C$ is number of $C$ invocations; and

$T_{SIM}$ is total simulation time

### 3.2.10 Service Performance Properties

Performance properties that Arcade can measure for services include the following:

- *Service Latency* - mean execution time of a service; expressed as follows:

$$L_{SVC} = \frac{\sum_{i=1}^{I_{SVC}} T_{SVC_i}}{I_{SVC}}$$

where:

$SVC$ is a service;

$T_{SVCi}$ is execution time of $SVC$ invocation $i$; and

$I_{SVC}$ is number of SVC invocations

- *Service Utilization* - amount of time spent performing a service; expressed as follows:

$$U_{SVC} = \frac{T_{SVC}}{T_{SIM}}$$

where:

$SVC$ is a service;

$T_{SVC}$ is total time spent executing $SVC$; and

$T_{SIM}$ is total simulation time

- *Service Throughput* - mean executions/unit time of a service; expressed as follows:

$$TP_{SVC} = \frac{I_{SVC}}{T_{SIM}}$$

where:

$SVC$ is a service;

$I_{SVC}$ is number of $SVC$ invocations; and

$T_{SIM}$ is total simulation time

#### *3.2.10.1 Translation: Arcade Metamodel to Simkernel*

Simpack provides a simulation kernel and simulation support routines, including support for definition and management of resources to perform work (called facilities in Simpack), event scheduling and delivery, statistical distribution sampling, and statistics collection. Simulation with Simpack is performed by moving tokens (units of work) through facilities (work performers).

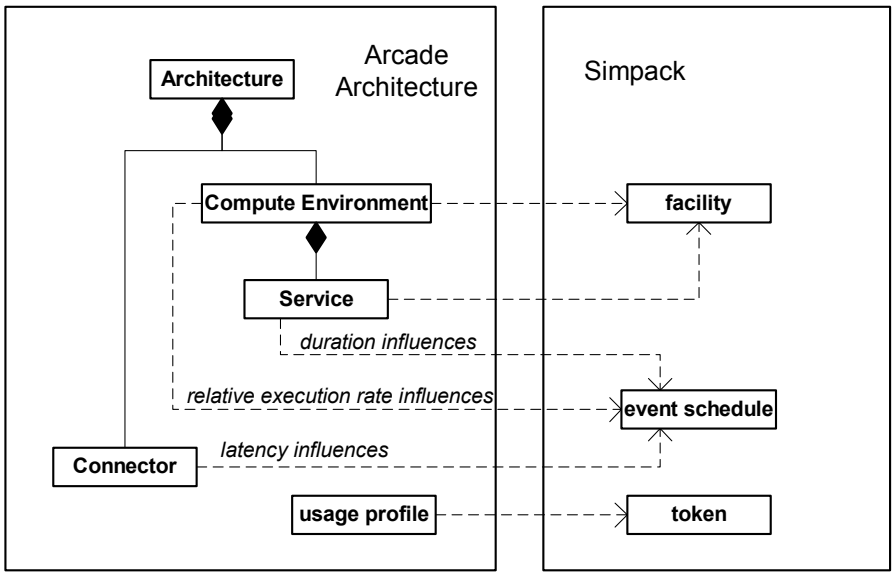


Figure 30 - Mapping From Arcade to Simpack

Figure 30 shows the mapping from the Arcade metamodel to elements of Simpack. Arcade maps services (Section 3.2.2.5) and compute environments (Section 3.2.2.9) to Simpack facilities, and maps domain usage profiles to Simpack tokens. Attributes of usage profiles, services, connectors, and compute

environments (such as probability of execution, duration, latency, etc.) are used to influence the scheduling of events during simulation.

Arcade defines simulation events and event handlers required to move usage profiles (tokens) through services (facilities) based on task sequences specified in usage profiles. Following this model, a usage profile must acquire the services that it needs to accomplish its task sequencing. Furthermore, a service must acquire the resources that it needs (e.g., its compute environment), and the duration of a service execution is affected by the duration as defined in the architecture as well as the relative execution speed of the compute environment. The scheduling of service executions is further conditioned by latencies associated with communication channels (Section 3.2.2.10). The level of concurrency in the simulation is controlled by the `IA_TIME` parameter which governs how often new usage profile tokens enter the system.

#### *3.2.10.2 Execution: Usage Profile Driven Simulation*

Arcade's performance simulation approach is illustrated in this section using the eDesign "Publish New Technical Document" usage profile (Chapter CHAPTER 4). As described in Section 3.2.10.1, the approach is based upon selecting and scheduling usage profiles for execution, and on providing a set of resources for the scheduled usage profiles to execute on.

A state diagram depicting the usage profile driven approach is shown in Figure 31. This diagram shows a composite state for the overall system execution, and a composite state for each usage profile that is executed. The

system execution state selects which usage profiles are executed, in what order they are executed, and the timing for starting usage profiles. Once a usage profile is started, the usage profile execution state controls the order of services to be executed and the duration of service executions. A usage profile execution terminates when there are no more services to execute for a given usage profile.

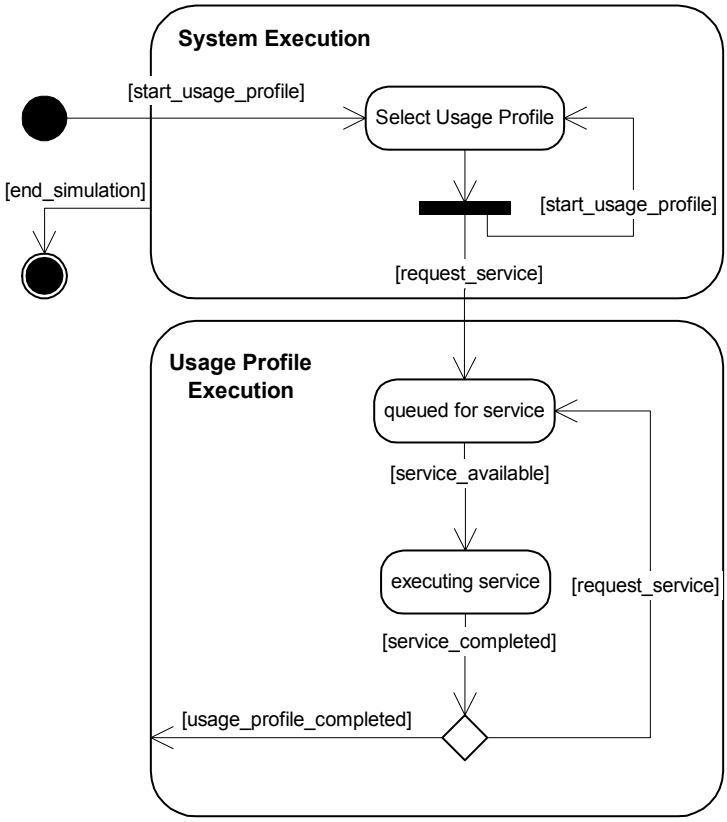


Figure 31 - State Model of Usage Profile Driven Simulation

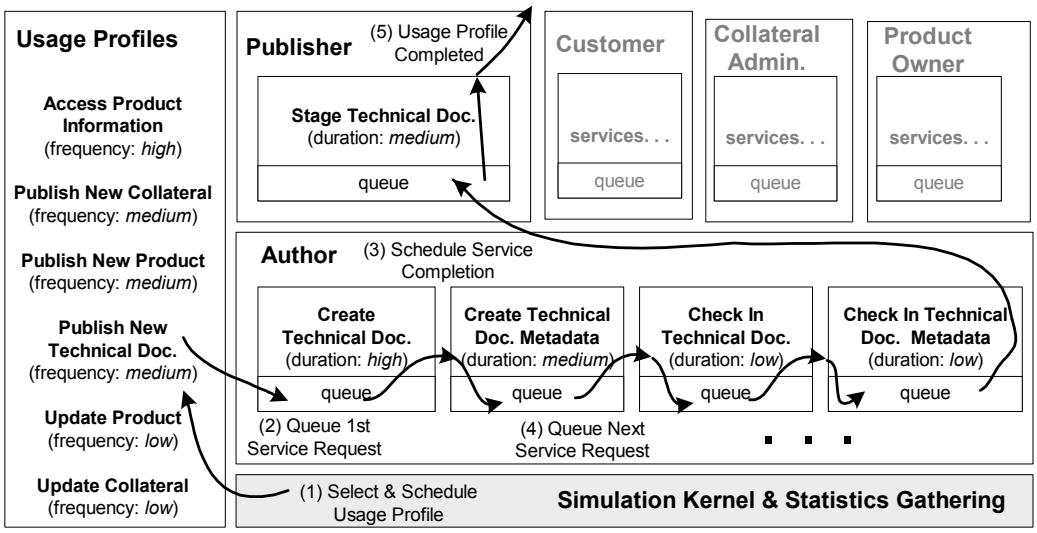


Figure 32 - Arcade Performance Simulation Approach

Figure 32 illustrates usage profile driven simulation with an example from the eDesign case study. Arcade initiates a simulation run by scheduling a *start_usage_profile* event using a negative exponential distribution to model the interarrival time of usage profile requests. When the Simpack kernel delivers a *start_usage_profile* event, Arcade randomly chooses a usage profile to begin executing (step 1 in Figure 32). This choice is weighted by the probability of execution associated with each usage profile (derived from stakeholder requirements regarding anticipated system usage). When a usage profile has been selected, Arcade schedules (1) a *request_service* event for the first service in the selected usage profile, and (2) another *start_usage_profile* event (using the appropriate interarrival time). The *request_service* event is queued pending service availability (step 2 in Figure 32). When the requested service becomes

available, Arcade allocates the facility associated with the requested service (causing subsequent service requests to become queued waiting for service completion) and schedules a *service_completed* event in the simulation using a normal distribution whose mean is a function of the service duration and the relative execution rate of the associated compute environment (step 3 in Figure 32). When the *service_completed* event is delivered, Arcade schedules a *request_service* event for the next service defined in the usage profile (step 4 in Figure 32). The timing of the *request_service* event is sampled from a normal distribution whose mean is a function of the channel latencies of exchanged events and data. This process is repeated until the usage profile has been completed (step 5 in Figure 32). Simulation continues in this manner until a maximum simulation time has been reached. Throughout this process, Arcade (with the aid of Simpack) collects performance statistics.

#### *3.2.10.3 Presentation: Graphing the Effects of System Loading*

Arcade presents the results of performance evaluations using line graphs. Stakeholders can select a performance property from a drop-down list, and Arcade will produce a graph that plots the mean value of the performance property over repeated simulation runs. Figure 33 shows an Arcade graph of component utilization ($U_C$) for an eDesign DRA version. This DRA version has five components (DRACs). Component utilization is plotted for each component over a series of simulation runs. The points on the graph are the mean component

utilization ($\overline{Uc}$) for ten simulation runs for each value of `IA_TIME = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 }` (in simulation time units).

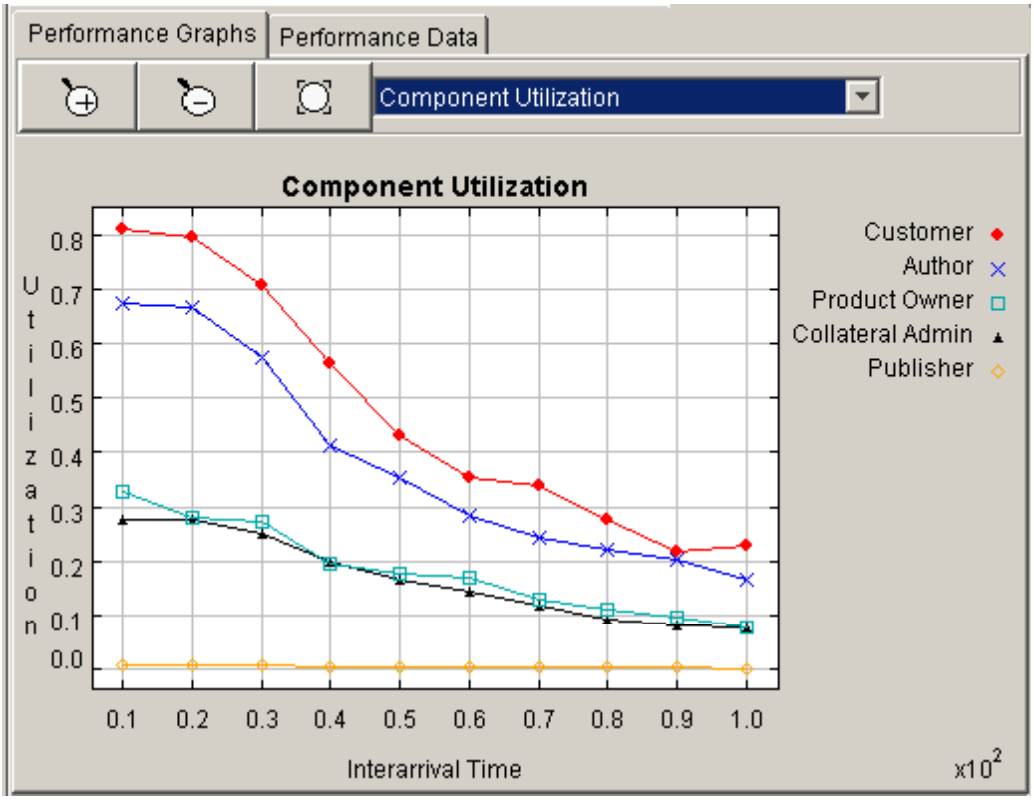


Figure 33 - Arcade Graph of Component Utilization

In Figure 33 it can be seen that, as the system becomes increasingly loaded, the mean component utilization of the "Customer" and "Author" DRACs begins to flatten out around `IA_TIME` = 20. Stakeholders could investigate possible causes of this effect by examining the values of other performance properties (around `IA_TIME` = 20).

### 3.2.11 Reliability Evaluation

Arcade employs an existing component-based reliability estimation technique called Scenario-Based Reliability Analysis (SBRA) [9, 117, 118]. This technique can be used to support Service-based reliability estimations by modeling each Service as a distinct component, and to support DRAC-based or TS-based reliability estimation by modeling each DRAC (or TS) as a component. The SBRA technique consists of (1) constructing a probabilistic reliability model called a Component Dependency Graph (CDG), and (2) applying the SBRA algorithm to the CDG model to yield a reliability estimate.

The following sections discuss the reliability property definition and the SBRA technique used by Arcade for reliability evaluation.

#### *3.2.11.1 Reliability Property Definition*

The reliability evaluation that Arcade performs is designed to estimate the conditional probability that an entire architecture will function correctly given the probability that a single architecture element will function correctly. An architecture element may be a service, a component (e.g., DRAC or TS), or a compute environment.

The following reliability properties can be evaluated by Arcade:

- *Service Reliability* - the reliability of the architecture with respect to the reliability of a service:

$$R_{SVC} = 1 - P(FAIL_{ARCH} \mid FAIL_{SVC})$$

where:

$R_{SVC}$ is the reliability of the architecture with respect to $SVC$; and

$SVC$ is a service in $ARCH$

- *Component Reliability* - the reliability of the architecture with respect to the reliability of a component (DRAC or TS):

$$R_C = 1 - P(FAIL_{ARCH} \mid FAIL_C)$$

where:

$R_C$ is the reliability of the architecture with respect to $C$; and

$C$ is a component in $ARCH$

- *Compute Environment Reliability* - the reliability of the architecture with respect to the reliability of a compute environment:

$$R_{CE} = 1 - P(FAIL_{ARCH} \mid FAIL_{CE})$$

where:

$R_{CE}$ is the reliability of the architecture with respect to $CE$; and

$CE$ is a compute environment in $ARCH$

The Arcade reliability evaluation approach estimates this property for all architecture elements in a class over a range of probabilities for architecture element failure. The SBRA reliability evaluation technique is described below.

#### *3.2.11.2 Scenario Based Reliability Analysis*

The SBRA reliability estimation technique has two steps: (1) constructing the Component Dependency Graph (CDG), and (2) applying the Scenario-Based Reliability Analysis (SBRA) algorithm to the CDG for reliability estimation. The graph model is described below. This is followed by a discussion of the SBRA algorithm, and how Arcade employs the algorithm for reliability evaluation.

The CDG model is a connected graph defined by:

$$CDG = <N, E, init, term>$$

where:

$<N, E>$ is a directed graph,

$init$ is the start node, $term$ is the termination node,

$N$ is the set of nodes in the graph, and

$E$ is the set of edges in the graph

A node ($n_i$) in the CDG models an architecture element (e.g., a service, a component (DRAC, or TS), or a compute environment), and is defined as:

$$n_i = <Elem_i, R_{Elemi}, AE_{Elemi}>$$

where:

$Elem_i$ is the architecture element name,

$R_{Elemi}$ is the reliability of $Elem_i$, and

$AE_{Elemi}$ is the average execution time of $Elem_i$

A directed edge ($e_{ij}$) in the CDG models the execution path from one architecture element to another, and is defined as:

$$e_{ij} = <T_{ij}, R_{Tij}, P_{Tij}>$$

where:

$T_{ij}$ is the transition name from node $n_i$ to node $n_j$,

$R_{Tij}$ is the transition reliability, and

$P_{Tij}$ is the transition probability

#### *3.2.11.3 Translation: Arcade Metamodel to CDG*

Arcade creates the CDG model by mapping architecture elements (e.g., services, components (DRACs or TSs), or compute environments) to CDG nodes, and by mapping edges from an Execution Space (Section 3.2.6.3) to CDG edges. The type of architecture element for which reliability is to be evaluated determines whether the CDG nodes represent components (DRACs or TSs), services, or compute environments. This mapping is shown in Figure 34.

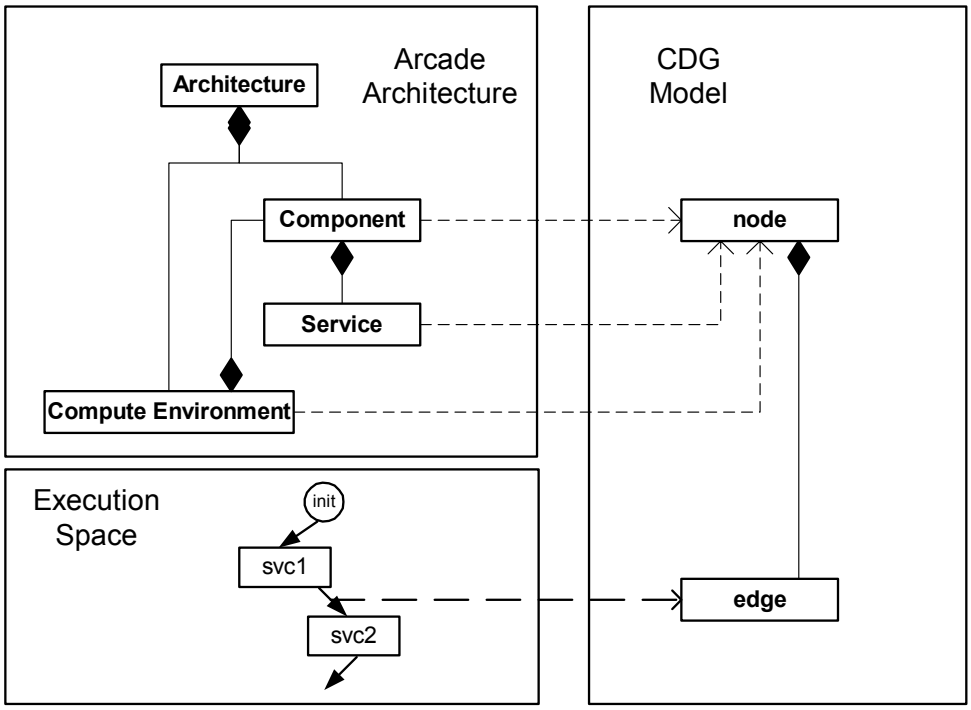


Figure 34 - Mapping the Arcade Model to a CDG

Construction of the CDG requires estimation of model parameters. The parameters, and how they are estimated for use with SBRA are described below.

- $P_{Ui}$ - Usage Profile Probability; usage profiles and their probability of execution are provided by stakeholders.

- $AE_{Elemi}$ - Average Execution Time of an architecture element; calculated using the following formula:

$$AE_{Elemi} = \sum_{k=1}^{|U|} P_{Uk} * Duration(Elem_i) \mid Elem_i \in U_k$$

where:

$P_{Uk}$ is the probability of execution of usage profile $U_k$, and

$Duration(Elem_i)$ is the duration of an execution of $Elem_i$ in usage profile $U_k$ as specified in the architecture.

- $P_{Tij}$ - Transition Probability of an edge; calculated as follows:

$$P_{Tij} = \sum_{k=1}^{|U|} P_{Uk} * \left( \frac{|Interact(Elem_i, Elem_j)|}{\sum_{l=1}^{N} |Interact(Elem_i, Elem_l)|} \right) \mid Elem_i,\ Elem_j \in U_k$$

where:

$P_{Uk}$ is the probability of execution of usage profile $U_k$,

$N$ is the number of architecture elements in the architecture, and

$|Interact(Elem_i, Elem_j)|$ is the number of "interactions" between $Elem_i$ and $Elem_j$ in usage profile $U_k$

Transition Probability is estimated using the service interactions in a usage profile and the usage profile's probability of execution. An "interaction" between two architecture elements, $Elem_i$ and $Elem_j$, occurs when execution moves from $Elem_i$ to $Elem_j$ in a usage profile.

- $AE_{ARCH}$ - Average Execution Time of the architecture; used to guarantee termination of the SBRA algorithm in the presence of cycles in usage profiles. $AE_{ARCH}$ is calculated using the following formula:

$$AE_{ARCH} = \sum_{k=1}^{|U|} P_{Uk} * Time(Uk)$$

where:

$P_{Uk}$ is the probability of execution of usage profile $U_k$, and

$Time(U_k)$ is the execution time of usage profile $U_k$ (calculated using duration information in the architecture).

The following section discusses the SBRA algorithm.

#### *3.2.11.4 Execution: The SBRA Algorithm*

The CDG model description above works equally well for evaluating reliability of various architecture elements (e.g., services, components (DRACs or TSs), and compute environments). Depending upon the type of reliability to be evaluated, the CDG model is built with nodes representing the type of architecture element to be evaluated. Once the CDG has been built, the SBRA algorithm can be applied.

The SBRA algorithm calculates reliability by iterating over the transitions in a CDG graph model. During iteration, the algorithm chooses paths based upon transition probabilities ($P_{Tij}$). As transitions are followed, cumulative reliability metrics are calculated using transition reliabilities ($R_{Tij}$) and architecture element reliabilities ($R_{Elemi}$). Execution time is accumulated using the average element execution time parameters ($AE_{Elemi}$). Iteration continues until the pre-calculated average system execution time ($AE_{ARCH}$) is reached.

Arcade's approach for reliability evaluation using SBRA involves repeatedly applying the SBRA algorithm to the CDG as follows: (1) select an architecture element $Elem_p$, (2) set the reliability of all architecture elements $Elem_q \neq Elem_p$ to 100%, and (3) vary the reliability parameter ($R_{Elemp}$) of $Elem_p$ over the range $0\% < R_{Elemp} \leq 100\%$ to assess the sensitivity of reliability of the entire architecture to the reliability of $Elem_p$. For each value used in step (3) the SBRA algorithm is applied to calculate the architecture reliability. This process is repeated for each element in the architecture. The result is a matrix of

reliability values that represents how reliability of the entire architecture is affected by the reliability of each architecture element.

The following section describes how results of this reliability evaluation process are presented to Arcade stakeholders.

#### *3.2.11.5 Presentation: Sensitivity Graphs*

Arcade presents reliability sensitivity matrices (calculated using the algorithm described in Section 3.2.11.4) as line graphs (Figure 35).

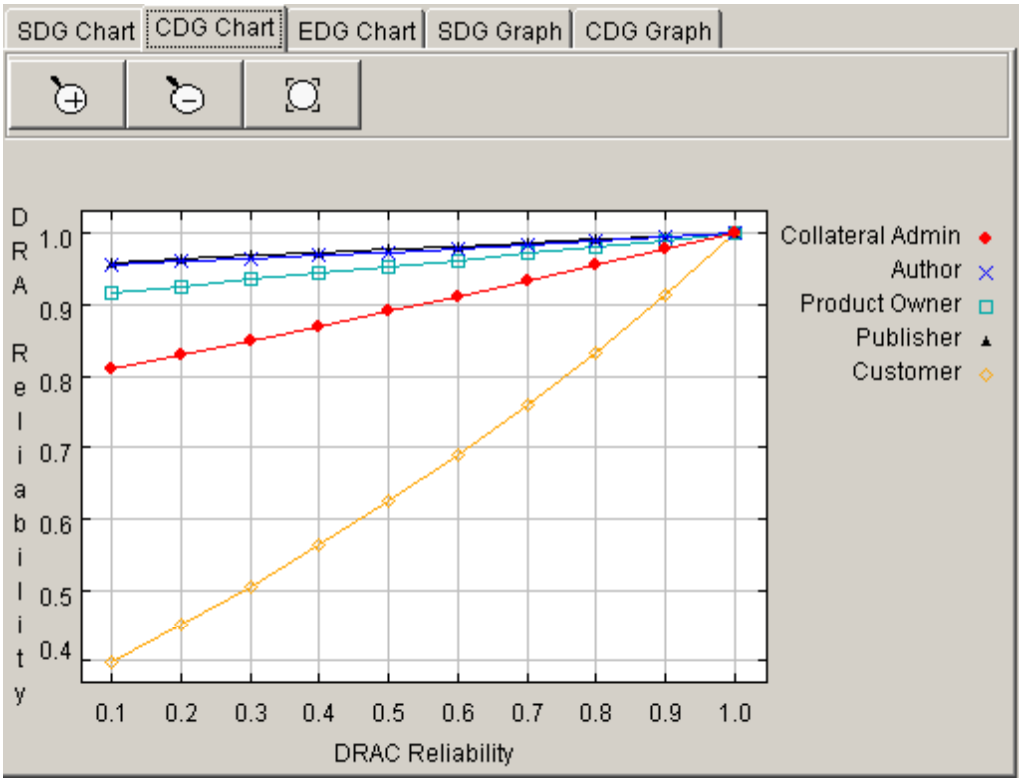


Figure 35 - Arcade Reliability Sensitivity graph

Reliability sensitivity graphs include a series for each architecture element that has been evaluated. The x-axis represents the value of $R_{Elemp}$ while the y-axis represents the resulting overall architecture reliability ($R_{ARCH}$). In this example it can be seen that the overall architecture reliability is most sensitive to the reliability of the "Customer" DRAC. Similar charts are available in Arcade for the other architectural elements that can be evaluated using reliability sensitivity evaluation.

The following section discusses architecture comparison techniques.

### 3.2.12 Architecture Comparison Techniques

The general process for Arcade architecture comparisons was depicted in Figure 12. This process requires that a set of metrics be defined by which architectures can be compared and ranked. This set of metrics is defined in the following sections for each property that Arcade can evaluate.

The architecture ranking technique is based upon defining a mechanism for detecting and reporting *property exceptions*. To accomplish this, a *threshold value* is defined for each property metric. A property metric that is outside of an acceptable range defined by its threshold value is reported as a property exception. The ranking technique then orders a set of architectures based upon the number of property exceptions that were reported for each architecture (where fewer property metric exceptions results in a higher rank).

The technique of ranking based upon property exceptions requires certain property metrics to be normalized. Normalization is important because many of the dynamic properties can have wide ranges of values for various architectural elements. In many cases these property metric values should actually be considered "normal". For example, it should not be an error when a service that is expected to have low utilization is actually measured to have low utilization. However, an error should be indicated when a measured property has an unexpected value. For dynamic properties that require normalization to ensure proper comparison, the metrics defined are a ratio between a measured value and an expected value. Where applicable, normalized metrics will be defined in the following sections.

### 3.2.13 Correctness Metrics and Exceptions

The metrics and associated exceptions used for correctness properties are defined as follows:

- *Safety Metric* - the number of safety errors detected; this metric is not required to be normalized because it is measured for the entire architecture:

$$M(C_S) = |safety\ errors|$$

where:

$|safety\ errors|$ is the number of safety errors

*Safety Metric Exceptions* - the rule used to detect a safety metric exception is defined as follows:

$$EXCP(C_S) = M(C_S) - T(C_S)$$

where:

$M(C_S)$ is the safety metric; and

$T(C_S)$ is the safety metric threshold

- *Liveness Metric* - the number of liveness errors detected; this metric is not required to be normalized because it is measured for the entire architecture:

$$M(C_L) = |liveness\ errors|$$

where:

$|liveness\ errors|$ is the number of liveness errors

*Liveness Metric Exceptions* - the rule used to detect a liveness metric exception is defined as follows:

$$EXCP(C_L) = M(C_L) - T(C_L)$$

where:

$M(C_L)$ is the liveness metric; and

$T(C_L)$ is the liveness metric threshold

### 3.2.14 Performance Metrics and Exceptions

Performance metrics are required to be normalized due to the wide range of values across various architectural elements. The general approach for normalization is to define metrics that are a ratio as follows:

- *Normalized Metric* - calculated as the ratio:

$$M(X) = \frac{EXP(X)}{X}$$

where:

$X$ is a performance property;

$EXP(X)$ is the expected value of $X$; and

$M(X)$ is the metric for $X$

Expected values for performance properties can be calculated from (1) static information contained within the architecture description and (2) the parameters used to control a performance simulation (Section 3.2.7). Sections 3.2.14.1 through 3.2.14.3 introduce the techniques for calculating expected values for performance properties used in architecture comparisons. This is followed by the definitions of performance property metrics and the rules used for reporting performance property exceptions in Sections 3.2.14.4 through 3.2.14.6

#### *3.2.14.1 Expected Values for Usage Profile Performance Properties*

The following expected values can be calculated for usage profile performance properties. These expected values depend solely upon static information contained in an architecture specification (Section 3.2.2) and the parameters used to set up a simulation run (Section 3.2.7).

- *Expected Usage Profile Latency* - the sum of durations specified in the architecture for all services composing the usage profile:

$$EXP(L_{UP}) = \sum D_{SVC} \mid SVC \in UP$$

where:

$UP$ is a usage profile;

$SVC$ is a service; and

$D_{SVC}$ is the duration of $SVC$ as specified in the architecture model

- *Expected Usage Profile Throughput* - the ratio of expected invocations of a usage profile to total simulation time:

$$EXP(TP_{UP}) = \frac{P(UP)\frac{SIM\_TIME}{IA\_TIME}}{SIM\_TIME}$$

where:

$UP$ is a usage profile;

*P(UP)* is the probability of executing *UP* as specified in the architecture model;

*IA_TIME* is the simulation interarrival time parameter; and

*SIM_TIME* is the simulation time parameter

#### *3.2.14.2 Expected Values for Component Performance Properties*

The following expected values can be calculated for component performance properties. These expected values depend solely upon static information contained in an architecture specification (Section 3.2.2) and the parameters used to set up a simulation run (Section 3.2.7).

- *Expected Component Utilization* - the ratio of expected time spent executing services in a component to the total simulation time:

$$EXP(U_C) = \frac{\sum \left( D_{SVC} * P(UP) \frac{SIM\_TIME}{IA\_TIME} \right)}{SIM\_TIME} \mid SVC \in C, SVC \in UP$$

where:

*C* is a component;

*SVC* is a service;

$D_{SVC}$ is duration of *SVC* as specified in the architecture model;

*P(UP)* is probability of executing *UP* as specified in the architecture;

*IA_TIME* is the interarrival time parameter; and

*SIM_TIME* is the simulation time parameter

- *Expected Component Throughput* - the ratio of expected invocations of a component to total simulation time:

$$EXP(TP_C) = \frac{\sum\left(P(UP)\frac{SIM\_TIME}{IA\_TIME}\right)}{SIM\_TIME} \mid SVC \in C, SVC \in UP$$

where:

`C` is a component;

`UP` is a usage profile;

`SVC` is a service;

`P(UP)` is probability of executing *UP* as specified in the architecture;

`IA_TIME` is the interarrival time parameter; and

`SIM_TIME` is the simulation time parameter

#### *3.2.14.3 Expected Values for Service Performance Properties*

The following expected values can be calculated for service performance properties. These expected values depend solely upon static information contained in an architecture specification (Section 3.2.2) and the parameters used to set up a simulation run (Section 3.2.7).

- *Expected Service Latency* - equivalent to the duration of a service as specified in the architecture:

$$EXP(L_{SVC}) = D_{SVC}$$

where:

$SVC$ is a service; and

$D_{SVC}$ is the duration of $SVC$ as specified in the architecture model

- *Expected Service Utilization* - the ratio of expected time spent executing a service to simulation time:

$$EXP(U_{SVC}) = \frac{\sum\left(D_{SVC} * P(UP)\frac{SIM\_TIME}{IA\_TIME}\right)}{SIM\_TIME} \mid SVC \in UP$$

where:

$SVC$ is a service;

$D_{SVC}$ is duration of $SVC$ as defined in the architecture model;

*P(UP)* is probability of executing UP as specified in the architecture model;

*IA_TIME* is the interarrival time parameter; and

*SIM_TIME* is the simulation time parameter

- *Expected Service Throughput* - the ratio of the expected number of invocations of a service to simulation time:

$$EXP(TP_{SVC}) = \frac{\sum\left(P(UP)\frac{SIM\_TIME}{IA\_TIME}\right)}{SIM\_TIME} \mid SVC \in UP$$

where:

*UP* is a usage profile;

*SVC* is a service;

*P(UP)* is probability of executing UP as specified in the architecture model;

*IA_TIME* is the interarrival time parameter; and

*SIM_TIME* is the simulation time parameter

#### *3.2.14.4 Usage Profile Property Metrics and Exceptions*

Given the usage profile performance property expected values defined in Section 3.2.14.1, the following usage profile performance metrics can be defined.

Each performance metric definition is accompanied by a definition of the rule used to detect a property exception (Section 3.2.7).

- *Usage Profile Latency Metric* - calculated as the ratio:

$$M(L_{UP}) = \frac{EXP(L_{UP})}{L_{UP}}$$

where:

$UP$ is a usage profile;

$EXP(L_{UP})$ is expected latency of $UP$; and

$L_{UP}$ is measured latency of $UP$

*Usage Profile Latency Exceptions* - the rule used to detect a usage profile latency metric exception is defined as follows:

$$\neg\left(1 - T(L_{UP}) < M(L_{UP}) < 1 + T(L_{UP})\right) \rightarrow EXCP(L_{UP})$$

where:

$UP$ is a usage profile;

$M(L_{UP})$ is the usage profile latency metric; and

$T(L_{UP})$ is the usage profile latency metric threshold

- *Usage Profile Throughput Metric* - calculated as the ratio:

$$M(TP_{UP}) = \frac{EXP(TP_{UP})}{TP_{UP}}$$

where:

$UP$ is a usage profile;

$EXP(TP_{UP})$ is expected throughput of $UP$;

$TP_{UP}$ is the measured throughput of $UP$

*Usage Profile Throughput Exceptions* - the rule used to detect a usage profile throughput metric exception is defined as follows:

$$\neg\left(1 - T(TP_{UP}) < M(TP_{UP}) < 1 + T(TP_{UP})\right) \rightarrow EXCP(TP_{UP})$$

where:

$UP$ is a usage profile;

$M(TP_{UP})$ is the usage profile throughput metric; and

$T(TP_{UP})$ is the usage profile throughput metric threshold

#### *3.2.14.5 Component Performance Property Metrics and Exceptions*

Given the component performance property expected values defined in Section 3.2.14.2, the following component performance metrics can be defined. Each performance metric definition is accompanied by a definition of the rule used to detect a property exception (Section 3.2.7).

- *Component Utilization Metric* - calculated as the ratio:

$$M(U_C) = \frac{EXP(U_C)}{U_C}$$

where:

$C$ is a component;

$EXP(U_C)$ is expected component utilization; and

$U_C$ is measured component utilization

*Component Utilization Exceptions* - the rule used to detect a component utilization metric exception is defined as follows:

$$\neg\left(1 - T(U_C) < M(U_C) < 1 + T(U_C)\right) \rightarrow EXCP(U_C)$$

where:

$C$ is a component;

$M(U_C)$ is the component utilization metric; and

$T(U_C)$ is the component utilization metric threshold

- *Component Throughput Metric* - calculated as the ratio:

$$M(TP_C) = \frac{EXP(TP_C)}{TP_C}$$

where:

$C$ is a component;

$EXP(TP_C)$ is expected component throughput; and

$TP_C$ is measured component throughput

*Component Throughput Exceptions* - the rule used to detect a component throughput metric exception is defined as follows:

$$\neg\left(1 - T(TP_C) < (M(TP_C) < 1 + T(TP_C)\right) \rightarrow EXCP(TP_C)$$

where:

$C$ is a component;

$M(TP_C)$ is the component throughput metric; and

$T(TP_C)$ is the component throughput metric threshold

#### *3.2.14.6 Service Performance Property Metrics and Exceptions*

Given the service performance property expected values defined in Section 3.2.14.3, the following component performance metrics can be defined. Each performance metric definition is accompanied by a definition of the rule used to detect a property exception (Section 3.2.7).

- *Service Latency Metric* - calculated as the ratio:

$$M(L_{SVC}) = \frac{EXP(L_{SVC})}{L_{SVC}}$$

where:

$SVC$ is a service;

$EXP(L_{SVC})$ is expected service latency; and

$L_{SVC}$ is measured service latency

*Service Latency Exceptions* - the rule used to detect a service latency metric exception is defined as follows:

$$\neg\left(1 - T(L_{SVC}) < M(L_{SVC}) < 1 + M(L_{SVC})\right) \rightarrow EXCP(L_{SVC})$$

where:

$SVC$ is a service;

$M(L_{SVC})$ is the service latency metric; and

$T(L_{SVC})$ is the service latency metric threshold

- *Service Utilization Metric* - calculated as the ratio:

$$M(U_{SVC}) = \frac{EXP(U_{SVC})}{U_{SVC}}$$

where:

$SVC$ is a service;

$EXP(U_{SVC})$ is expected service utilization; and

$U_{SVC}$ is measured service utilization

*Service Utilization Exceptions* - the rule used to detect a service utilization metric exception is defined as follows:

$$\neg\left(1 - T(U_{SVC}) < M(U_{SVC}) < 1 + T(U_{SVC})\right) \rightarrow EXCP(U_{SVC})$$

where:

$SVC$ is a service;

$M(U_{SVC})$ is the service utilization metric; and

$T(U_{SVC})$ is the service utilization metric threshold

- *Service Throughput Metric* - calculated as the ratio:

$$M(TP_{SVC}) = \frac{EXP(TP_{SVC})}{TP_{SVC}}$$

where:

$SVC$ is a service;

$EXP(TP_{SVC})$ is expected throughput of $SVC$;

$TP_{SVC}$ is the measured throughput of $SVC$

*Service Throughput Exceptions* - the rule used to detect a service throughput metric exception is defined as follows:

$$\neg\left(1 - T(TP_{SVC}) < M(TP_{SVC}) < 1 + T(TP_{SVC})\right) \rightarrow EXCP(TP_{SVC})$$

where:

$SVC$ is a service;

$M(TP_{SVC})$ is the service throughput metric; and

$T(TP_{SVC})$ is the service throughput metric threshold

### 3.2.15 Reliability Metrics and Exceptions

The metrics and associated exceptions used for reliability evaluation are defined as follows:

- *Service Reliability Metric* - the probability that the architecture will fail given that a service fails:

$$M(R_{SVC}) = 1 - P(FAIL_{ARCH} \mid FAIL_{SVC})$$

where:

$SVC$ is a service

*Service Reliability Metric Exceptions* - the rule used to detect a service reliability metric exception is defined as follows:

$$\left(M(R_{SVC}) < 1 - T(R_{SVC})\right) \rightarrow EXCP(R_{SVC})$$

where:

$SVC$ is a service;

$M(R_{SVC})$ is the service reliability metric; and

$T(R_{SVC})$ is the service reliability metric threshold

- *Component Reliability Metric* - the probability that the architecture will fail given that a component (DRAC or TS) fails:

$$M(R_C) = 1 - P(FAIL_{ARCH} \mid FAIL_C)$$

where:

$C$ is a component

*Component Reliability Metric Exceptions* - the rule used to detect a component reliability metric exception is defined as follows:

$$\left(M(R_C) < 1 - T(R_C)\right) \rightarrow EXCP(R_C)$$

where:

$C$ is a component;

$M(R_C)$ is the component reliability metric; and

$T(R_C)$ is the component reliability metric threshold

- *Compute Environment Reliability Metric* - the probability that the architecture will fail given that a compute environment fails:

$$M(R_{CE}) = 1 - P(FAIL_{ARCH} \mid FAIL_{CE})$$

where:

$CE$ is a compute environment

*Compute Environment Reliability Metric Exceptions* - the rule used to detect a compute environment reliability metric exception is defined as follows:

$$\left(M(R_{CE}) < 1 - T(R_{CE})\right) \rightarrow EXCP(R_{CE})$$

where:

$CE$ is a compute environment;

$M(R_{CE})$ is the compute environment reliability metric; and

$T(R_{CE})$ is the compute environment reliability metric threshold

### 3.2.16 Architecture Ranking Technique

The Arcade architecture ranking technique is used to compare the dynamic properties of a set of architectures. This technique is a constituent part of the Arcade comparison process described in Section 3.1.3. This technique ranks architectures based upon the number of property exceptions that are reported from the results of Arcade evaluations using the metrics and exceptions defined in Section 3.2.13, Section 3.2.14, and Section 3.2.15. For a set of architectures and a set of dynamic properties, the rankings are computed by averaging the ranks for individual properties within a property class (for example, *Usage Profile Latency* - $L_{UP}$ - is a property within the property class *Performance Properties*), and then summing the ranks by property class. This technique is expressed formally as follows:

$$RANK(ARCH) = \sum_{K} \overline{RANK(p)} \mid K \in \{C, P, R\}, p \in K$$

where:

$C$ is the set of *Correctness* properties evaluated;

$P$ is the set of *Performance* properties evaluated;

$R$ is the set of *Reliability* properties evaluated; and

$RANK$ is the ordinal rank of a property metric value for $ARCH$ in a set of property metric values for other architectures $\neq ARCH$

The typical representation of this ranking technique is a bar graph or a stacked bar graph. Examples of these types of graphs appear frequently in Chapter 5 and APPENDIX D.

# CHAPTER 4 THE eDESIGN CASE STUDY

This chapter describes a case study that was performed in association with the research and experimentation described in Chapter 3 and Chapter CHAPTER 5. The topics covered include how the Arcade tool and evaluation processes were employed for this case study and what results were obtained. The conclusions at the end of this chapter summarize the findings of the case study.

## 4.1 THE MOTOROLA eDESIGN SYSTEM

The eDesign system is being developed by Motorola's Semiconductor Products Sector (SPS) to efficiently deliver SPS product technical information and collateral products to internal and external Motorola customers. Major functionality in the eDesign system includes Document Authoring, Document Configuration Management, Content Administration, and Content Delivery. eDesign is being deployed with a mix of off-the-shelf as well as in-house developed applications. The eDesign system is intended to satisfy stakeholder requirements for viewing and downloading various product information, including: (1) product parametric data, (2) product configurations, and (3) collateral products and information available to support design and manufacturing of new products that incorporate Motorola SPS products into their design. The major areas of eDesign functionality and their associated stakeholders are illustrated in Figure 36.

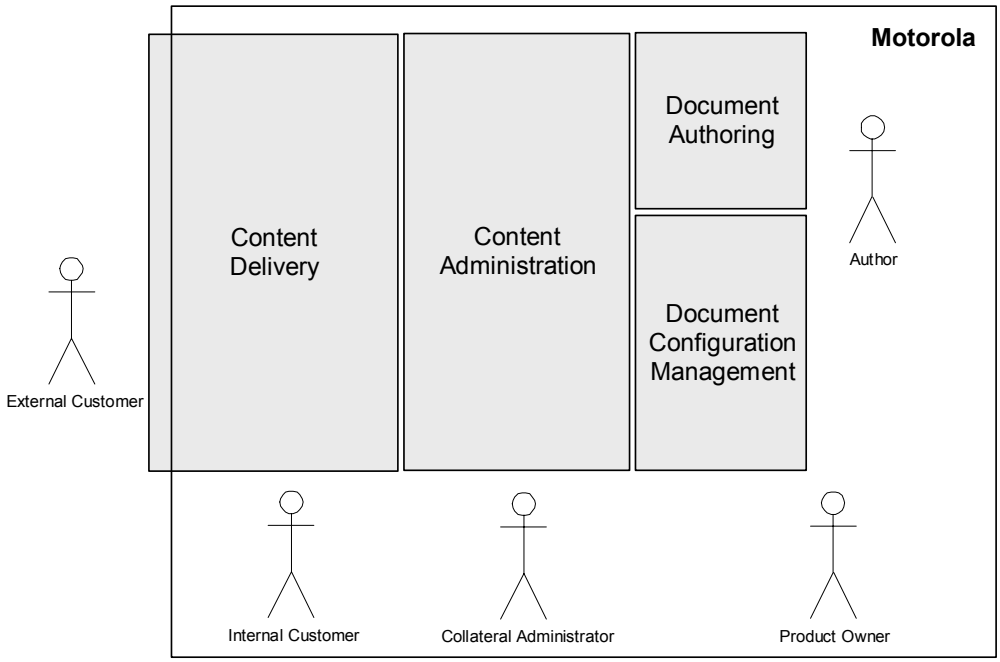


Figure 36 - Functionality and Stakeholders of the eDesign Domain

The eDesign project was well suited to the SEPA methodology and its supporting tools since it is being developed in a very iterative, rapid fashion (thus it could benefit from SEPA's approach to requirements evolution). Furthermore, in the e-business domain the business needs (e.g., domain requirements) are evolving at a less rapid pace than technology requirements (e.g., the AA and IA). Therefore, the goals of this effort were: (1) to provide the eDesign team with a requirements foundation upon which to rapidly evolve their system, (2) to characterize the dynamic properties governed by requirements so that the eDesign team can make better decisions related to evolving technology requirements, and (3) to provide system implementation guidance.

The following section discusses the overall approach used for the eDesign case study.

## 4.2 APPROACH

The first step in eliciting the domain requirements of the eDesign system was to perform Requirements Acquisition (RA) to acquire domain usage profiles describing functionality in the domain. Usage profiles help to explicitly define and consequently scope the domain functionality, data, timing, interactions, and user types. The full set of usage profiles (Table 1) acquired during RA sessions provided the basis from which an initial DRA version was derived.

| *Usage Profile* | *Frequency* | *Tasks* | *Performer* | *Duration* |
|---|---|---|---|---|
| **Publish New Collateral** | medium | Add Collateral File | Collateral Admin | high |
| | | Launch Collateral | Collateral Admin | low |
| **Publish New Product** | medium | Create Product Summary Page | Product Owner | high |
| | | Associate Collateral to PSP | Product Owner | med |
| | | Launch PSP | Product Owner | low |
| | | Associate PSP to a Category | Product Owner | low |
| **Access Product Information** | high | Search for Product | Customer | med |
| | | Select PSP | Customer | low |
| | | Download Product Collateral | Customer | high |
| **Publish New Technical Document** | medium | Create Technical Documentation | Author | high |
| | | Create Technical Document Metadata | Author | high |
| | | Check In Technical Documentation | Author | low |
| | | Check In Technical Document Metadata | Author | low |
| | | Stage Technical Documentation | Publisher | med |
| **Update Collateral** | low | Stage Collateral | Collateral Admin | low |
| | | Update Collateral Attributes | Collateral Admin | high |
| | | Launch Collateral | Collateral Admin | low |
| **Update Product** | low | Stage Product | Product Owner | low |
| | | Update Product | Product Owner | high |
| | | Launch Product | Product Owner | low |

Table 1 - eDesign Usage Profiles

A usage profile is composed of one or more domain tasks, where a task specification includes the name of a performer capable of executing the task, input data/events required for execution, output data/events produced, and pre-/post-conditions defining necessary conditions to begin execution and expected conditions following execution, respectively. Figure 37 shows the "Publish New Technical Document" usage profile in more detail.

**USAGE PROFILE**: Publish New Technical Document

**Task**: Create Technical Documentation
**Performer**: Author
**Input Data/Events**: Product Specification
**Output Data/Events**: Technical Documentation
**Pre-condition**: Request for Technical Documentation
**Post-condition**: Technical Document Created

**Task**: Create Technical Document Metadata
**Performer**: Author
**Input Data/Events**: document attributes, parametric data
**Output Data/Events**: Technical Document Metadata
**Pre-condition**: Technical Document Created
**Post-condition**: Technical Document Metadata Created

**Task**: Check In Technical Documentation
**Performer**: Author
**Input Data/Events**: Technical Documentation
**Output Data/Events**: none
**Pre-condition**: Technical Document Created
**Post-condi**tion: Technical Document Under Configuration Management

**Task**: Check In Technical Document Metadata
**Performer**: Author
**Input Data/Events**: Technical Document Metadata
**Output Data/Events**: none
**Pre-condition**: Technical Document Metadata Created
**Post-condition**: Technical Document Metadata Checked In

**Task**: Change Product State
**Performer**: Publisher
**Input Data/Events**: Technical Document, Technical Document Metadata
**Output Data/Events**: none
**Pre-condition**: Technical Document Checked In -and- Technical Document Metadata Checked In
**Post-condition**: Technical Document State is 'staged'

Figure 37 - The "Publish New Technical Document" Usage Profile

Additional requirements were gathered to support the definition of the eDesign AA and IA models. Following construction of the initial eDesign DRA versions, the team proceeded to evaluate its dynamic properties using Arcade. Several revisions were made to the eDesign requirements as a result of errors uncovered via Arcade evaluations. Following this, evaluations were subsequently performed for the eDesign AA and the eDesign IA.

## 4.3 DRA SPECIFICATION AND EVALUATION

The final eDesign DRA version illustrated in Figure 38 was derived from the usage profiles in Table 1.

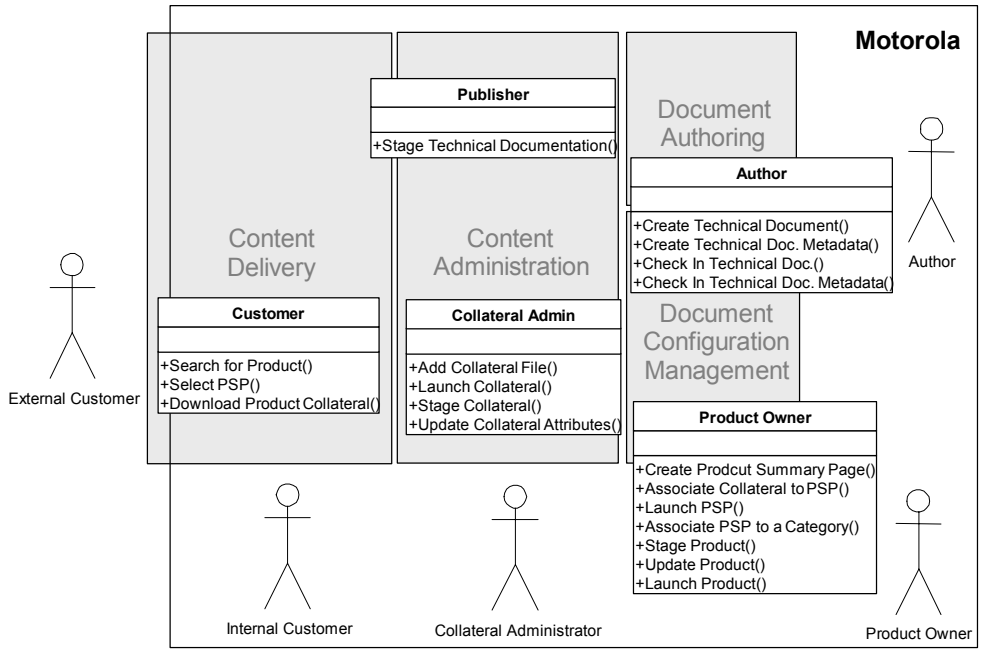


Figure 38 - eDesign Domain Reference Architecture

The DRA version in Figure 38 contains five DRACs representing domain performer roles identified by stakeholders: “Customer”, “Publisher”, “Collateral Admin”, “Author”, and “Product Owner”. Services were allocated to DRACs using the allocations of domain functionality to performers that were specified in domain usage profiles.

The following sections discuss the dynamic property evaluations that were performed for the eDesign DRA.

### 4.3.1 DRA Correctness Evaluation

Arcade evaluation detected a number of correctness errors in the initial eDesign DRA version. These errors were partially due to requirements acquisition and modeling errors, and partially due to the fact that correctness errors tend to propagate under model checking and simulation.

The types of requirements acquisition and modeling errors that occurred fell into two classes: (1) misinterpretation errors, and (2) specification errors. Misinterpretation errors occurred when information acquired from stakeholders and domain experts was incorrectly understood during requirements modeling. Specification errors occurred when mistakes were made transcoding unstructured requirements into requirements models.

Experience with eDesign shows that error propagation can artificially inflate the number and types of errors that are detected under dynamic property evaluations. For example, a liveness error associated with a particular service may cause many other liveness errors for other services that have causal

dependencies on that service (e.g., the dependent services require events or data to become available via the post-condition associated with the service exhibiting the initial liveness error). Propagation of correctness errors made it difficult for eDesign stakeholders to determine exactly which errors to address first. To mitigate this complexity issue, it was decided that correctness evaluations of DRA versions containing partial models of the eDesign domain would be appropriate. This decision allowed evaluation to focus on requirements errors while minimizing the effort to sort through correctness error propagations.

The technique employed to form partial models was to scope requirements modeled in a set of DRA versions according to usage profile boundaries. This resulted in a number of DRA versions, each of which was self-contained in terms of the ability to support execution of a single usage profile. The DRA versions and associated eDesign usage profiles are listed in Table 2.

| USAGE PROFILE | DRA VERSION |
|---|---|
| Publish New Collateral | $DRA_{UP1}$ |
| Publish New Product | $DRA_{UP2}$ |
| Access Product Information | $DRA_{UP3}$ |
| Publish New Technical Document | $DRA_{UP4}$ |
| Update Collateral | $DRA_{UP5}$ |
| Update Product | $DRA_{UP6}$ |

Table 2 - eDesign Partial Models by DRA Version

Arriving at a satisfactory full DRA version required several iterations of partial DRA version evaluations and corresponding requirements revisions. The number and types of errors detected, and their associated types of corrections are summarized by requirements revision in Figure 39 (REV0 was the initial version).

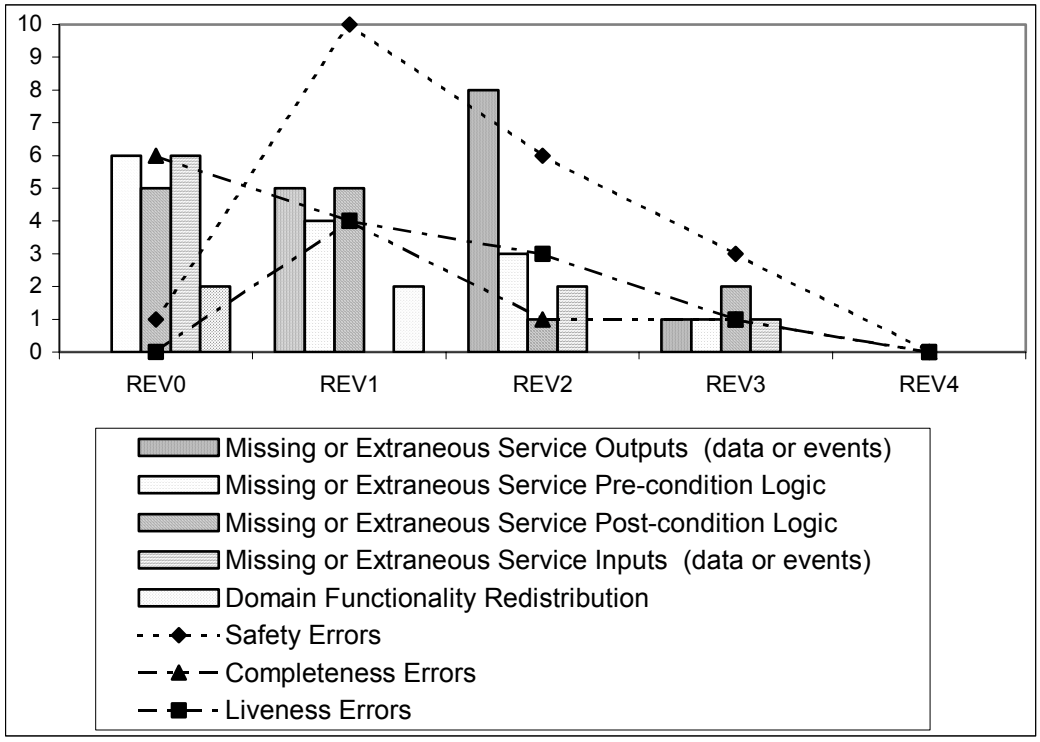


Figure 39 - eDesign Correctness Errors and Resolutions

A more detailed breakdown of correctness errors by DRA version is shown in Table 3. In this table, a DRA version is identified by its requirements revision level (where REV0 is the initial version), and the usage profile it is associated with (Table 2). For example, $DRA_{REV0,UP1}$ is a partial DRA version for requirements revision 0 and the "Publish New Collateral" usage profile.

| DRA VERSIONS | SAFETY ERRORS | LIVENESS ERRORS | COMPLETENESS ERRORS |
|---|---|---|---|
| $DRA_{REV0,UP1}$ | 0 | 0 | 2 |
| $DRA_{REV1,UP1}$ | 2 | 1 | 2 |
| $DRA_{REV2,UP1}$ | 0 | 0 | 0 |
| $DRA_{REV3,UP1}$ | 0 | 0 | 0 |
| $DRA_{REV4,UP1}$ | 0 | 0 | 0 |
| $DRA_{REV0,UP2}$ | 0 | 0 | 0 |
| $DRA_{REV1,UP2}$ | 5 | 3 | 1 |
| $DRA_{REV2,UP2}$ | 5 | 3 | 1 |
| $DRA_{REV3,UP2}$ | 2 | 1 | 1 |
| $DRA_{REV4,UP2}$ | 0 | 0 | 0 |
| $DRA_{REV0,UP3}$ | 1 | 0 | 0 |
| $DRA_{REV1,UP3}$ | 1 | 0 | 1 |
| $DRA_{REV2,UP3}$ | 1 | 0 | 0 |
| $DRA_{REV3,UP3}$ | 1 | 0 | 0 |
| $DRA_{REV4,UP3}$ | 0 | 0 | 0 |
| $DRA_{REV0,UP4}$ | 0 | 0 | 0 |
| $DRA_{REV1,UP4}$ | 0 | 0 | 0 |
| $DRA_{REV2,UP4}$ | 0 | 0 | 0 |
| $DRA_{REV3,UP4}$ | 0 | 0 | 0 |
| $DRA_{REV4,UP4}$ | 0 | 0 | 0 |
| $DRA_{REV0,UP5}$ | 0 | 0 | 2 |
| $DRA_{REV1,UP5}$ | 1 | 0 | 0 |
| $DRA_{REV2,UP5}$ | 0 | 0 | 0 |
| $DRA_{REV3,UP5}$ | 0 | 0 | 0 |
| $DRA_{REV4,UP5}$ | 0 | 0 | 0 |
| $DRA_{REV0,UP6}$ | 0 | 0 | 2 |
| $DRA_{REV1,UP6}$ | 1 | 0 | 0 |
| $DRA_{REV2,UP6}$ | 0 | 0 | 0 |
| $DRA_{REV3,UP6}$ | 0 | 0 | 0 |
| $DRA_{REV4,UP6}$ | 0 | 0 | 0 |
| TOTALS | 20 | 8 | 12 |

Table 3 - Correctness Errors by DRA Version

Table 4 classifies the types of errors detected as either misinterpretation errors or specification errors, and summarizes the types of corrections required for each class of error.

| TYPE OF ERROR | TYPE OF CORRECTION | % CORRECTIONS | % ERRORS |
|---|---|---|---|
| Misinterpretation | | | 43% |
| | output correction | 26% | |
| | input correction | 17% | |
| Specification | | | 57% |
| | pre-condition correction | 26% | |
| | post-condition correction | 24% | |
| | functionality redistribution | 7% | |

Table 4 - Summary of DRA Corrections

Completeness errors comprised the majority of errors uncovered by the earliest correctness evaluations. These initial completeness errors were addressed by adjusting pre- and post-conditions, and in one instance by distributing the domain functionality of a single DRA service across multiple services. Subsequently, safety and liveness errors began to be identified in greater numbers. The remaining iterations of evaluation and correction successively reduced the number of errors detected, and the majority of errors detected in the final iterations were safety errors. By the later iterations most of the required domain functionality was correctly modeled (e.g., there were fewer completeness and liveness errors), but there remained some corrections to be made with regard to

the details of data and event exchanges (e.g., most of the remaining errors were safety errors).

During the iterative evaluation and correction process, completeness errors were detected by stakeholder examinations of Execution Spaces produced by Arcade simulations (Section 3.2.6.2), and safety and liveness errors were detected using Arcade's automated model checking (Section 3.2.4, and Section 3.2.5). Table 5 categorizes the types of correctness errors found and the methods of detection for each revision of the eDesign requirements. These statistics highlight that simulation was an effective means of detecting correctness errors early in the process, and model checking was effective later in the process.

| REQUIREMENTS REVISION | SAFETY ERRORS | LIVENESS ERRORS | COMPLETENESS ERRORS | | DETECTED BY SIMULATION | DETECTED BY MODEL CHECKING |
|---|---|---|---|---|---|---|
| REV0 | 1 | 0 | 6 | | 6 | 1 |
| REV1 | 10 | 4 | 4 | | 4 | 14 |
| REV2 | 6 | 3 | 1 | | 1 | 9 |
| REV3 | 3 | 1 | 1 | | 1 | 4 |
| REV4 | 0 | 0 | 0 | | 0 | 0 |
| | | | | | | |
| totals | 20 | 8 | 12 | | 12 | 28 |

Table 5 - Summary Statistics for eDesign Correctness Evaluation

In summary, the steps used to specify the eDesign DRA were: (1) to iteratively evaluate and repair correctness errors for each of the partial models, (2) to merge the correct partial DRA versions into a unified DRA version, and (3) to evaluate the correctness of the resulting DRA. The merging of the partial DRA

versions into a full DRA version (e.g., steps 2 and 3) went smoothly with no additional correctness errors identified in the merged DRA.

### 4.3.2 DRA Performance Evaluation

The SEPA Domain Reference Architecture (DRA) encompasses performance characteristics inherent to the domain by specifying in a technology independent fashion (1) what communication is necessary among services, and (2) a recommended allocation of services to Domain Reference Architecture Components (DRACs). Therefore, Arcade DRA performance evaluations are focused on understanding domain performance characteristics and identifying performance-critical aspects of domain requirements. Stakeholders can also reuse DRA performance evaluation results during specification of the AA and IA (Section 1.2.1). This reuse allows later performance evaluations to focus on an improved understanding of previously identified performance-critical aspects of the system.

The eDesign stakeholders indicated that their primary concern for performance was the system response time associated with finding and downloading information from the eDesign web site (e.g., the "Access Product Information" usage profile in Table 1). Accordingly the DRA performance evaluation results described below highlight the performance evaluation associated with the "Access Product Information" usage profile.

Results of early performance evaluations of the eDesign DRA indicated that the "Access Product Information" usage profile was likely to suffer unacceptable latencies (e.g., response times) as the system became increasingly loaded. This is illustrated in Figure 40, where the latency of the "Access Product Information" usage profile can be seen to increase as the system becomes loaded (shorter usage profile request interarrival times represent increased system load).

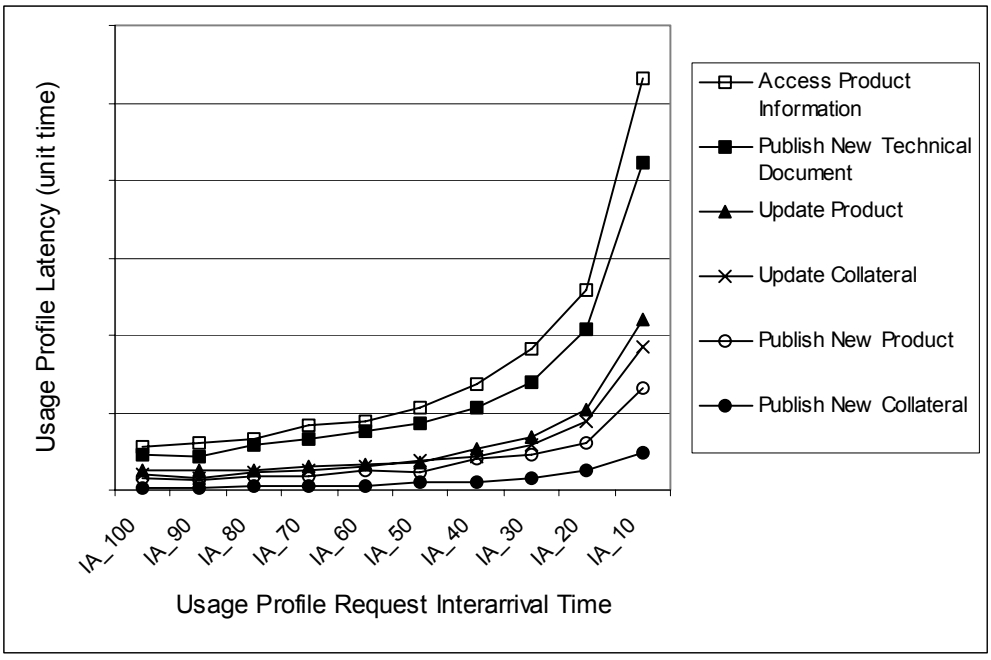


Figure 40 - eDesign DRA Usage Profile Latencies

Figure 41 shows average service utilization percentages under heavy system loading (for this discussion, defined as usage profile request interarrival times of less than 40 simulation time units). This graph highlights that the utilization of the "Download Product Collateral" service (a constituent service of the "Access Product Information" usage profile) was approximately100 percent under heavy system loads, indicating that the "Download Product Collateral" service was the most performance-critical service in the system.

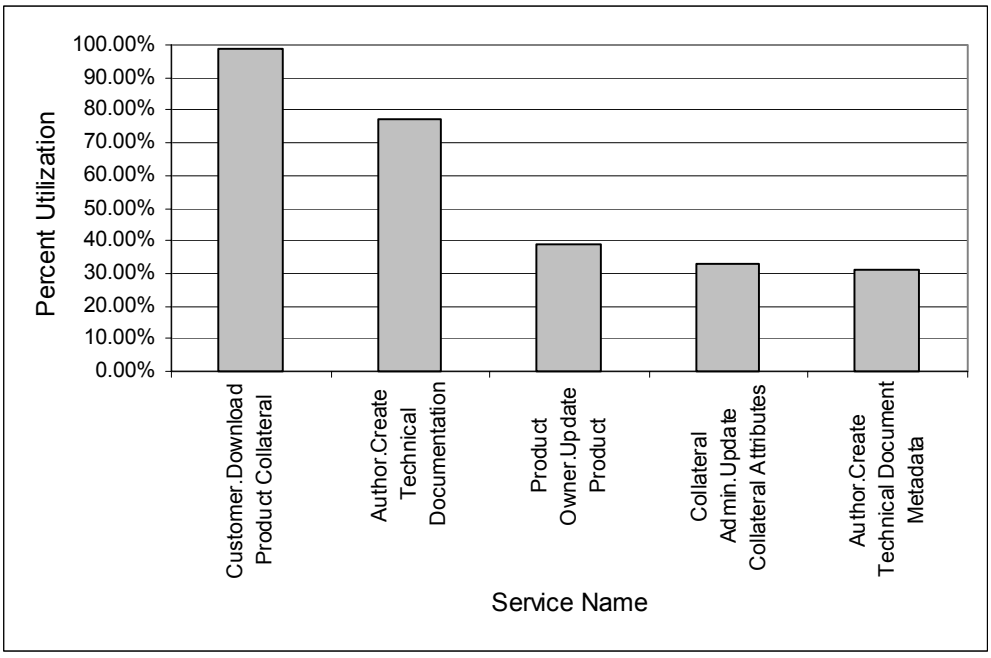


Figure 41 - eDesign DRA Service Utilizations

Based upon stakeholder priorities and the insights gained from DRA performance evaluation, it was determined that AA performance evaluation

should continue to focus on the critical "Access Product Information" usage profile and its constituent "Download Product Collateral" service. The eDesign AA performance evaluation is described in Section 4.4.1.

### 4.3.3 DRA Reliability Evaluation

Arcade can evaluate two reliability properties for a DRA. These two properties are $R_{SVC}$ - the reliability of the DRA with respect to the reliability of services, and $R_C$ - the reliability of the DRA with respect to the reliability of DRACs (Section 3.2.11.1). Both of these properties were evaluated for the eDesign DRA.

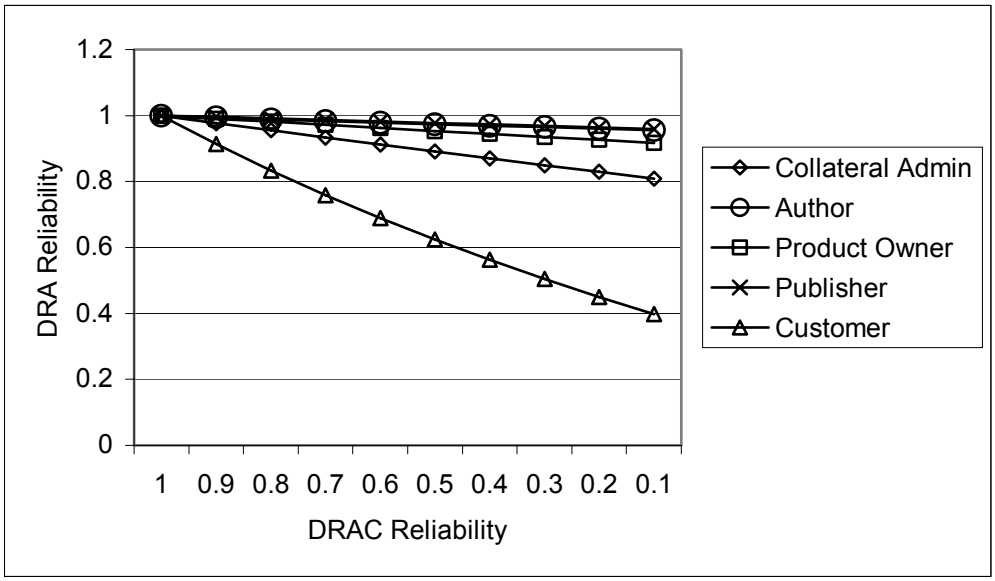


Figure 42 - Arcade Reliability Evaluation for eDesign DRA ($R_C$)

The results of the $R_C$ evaluation for the eDesign DRA are shown in Figure 42. Here it can be seen that the DRA is most sensitive to the reliability of the "Customer" DRAC. The identification of the "Customer" DRAC as critical in both performance and reliability evaluations was noted by eDesign stakeholders as reason to pay particular attention to the implementation of domain functionality associated with that DRAC.

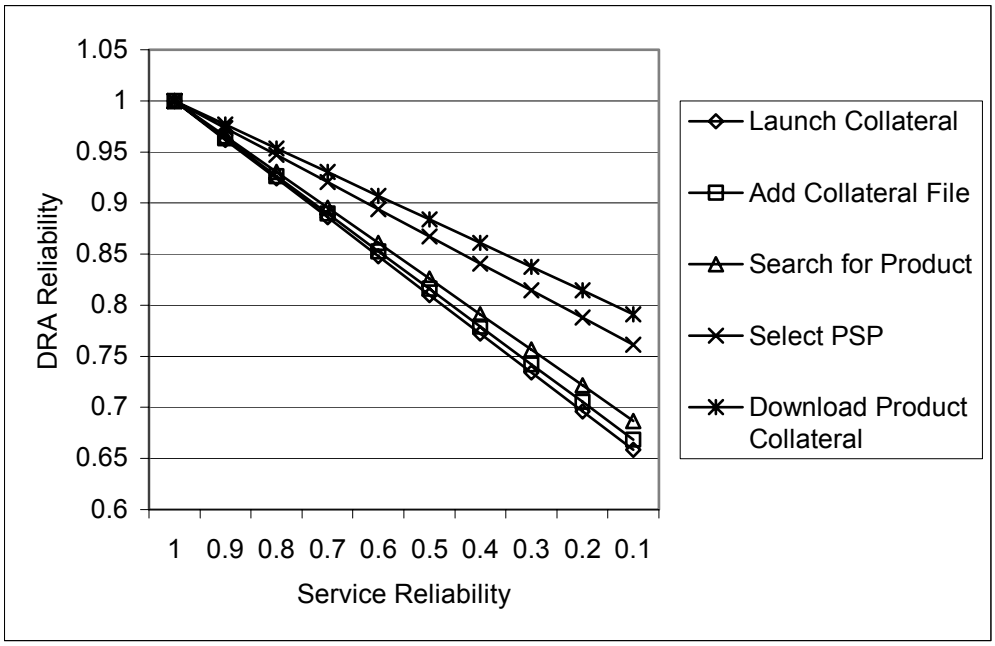


Figure 43 - Arcade Reliability Evaluation for eDesign DRA ($R_{SVC}$)

Figure 43 shows the reliability graph for $R_{SVC}$ (service reliability) for five services that eDesign DRA reliability evaluation identified as the services to which overall architecture reliability was most sensitive. Again the "Download

Product Collateral" service is identified as a critical service for reliability (in addition to its being identified as critical to performance Section 4.3.2). Two additional services from the "Customer" DRAC were identified as reliability critical: "Search For Product" and "Select PSP". It was also noted that two services associated with the "Collateral Admin" DRAC (e.g., "Launch Collateral" and "Add Collateral File") were among the services to which the architecture reliability was most sensitive. It can be seen in Figure 42 that the sensitivity of the DRA to the reliability of the "Collateral Admin" DRAC was second only to that of the "Customer" DRAC. Thus it was determined that the domain functionality associated with the "Customer" and "Collateral Admin" DRACs was critical with respect to system reliability, and this functionality should be evaluated with particular attention following across-architecture evolution introduced by the AA.

## 4.4 AA SPECIFICATION AND EVALUATION

The SEPA Application Architecture (AA) provides a framework for specifying both functional and non-functional application requirements, including, but not limited to, application look-and-feel and runtime performance requirements (Section 2.2.2). The AA is formed when Technology Solution (TS) instances (e.g., Applications, solutions implemented, under development, or envisioned) are selected to fulfill services (domain functionality) specified in the DRA. The selected TS instances must meet implementation-related constraints imposed on services as specified by stakeholders' application requirements. The

relationship between the AA and the DRA is represented by a Domain Map. A Domain Map is comprised of a set of Attribute Maps and Service Maps that express the AA to DRA relation through links to the domain attributes (data) and services (functions) satisfied by a TS instance in the AA.

The eDesign AA consists of five TS instances: “Product Catalog”, “CAT”, “Frame”, “Word”, and “Blade”. Three of these TS instances are commercial off-the-shelf (COTS) software components selected to provide required DRA functionality (“Frame”, “Word”, and “Blade”), while two TS instances were developed in-house at Motorola (“Product Catalog”, and “CAT”). Figure 44 illustrates services from the eDesign DRA mapped onto these TS instances.

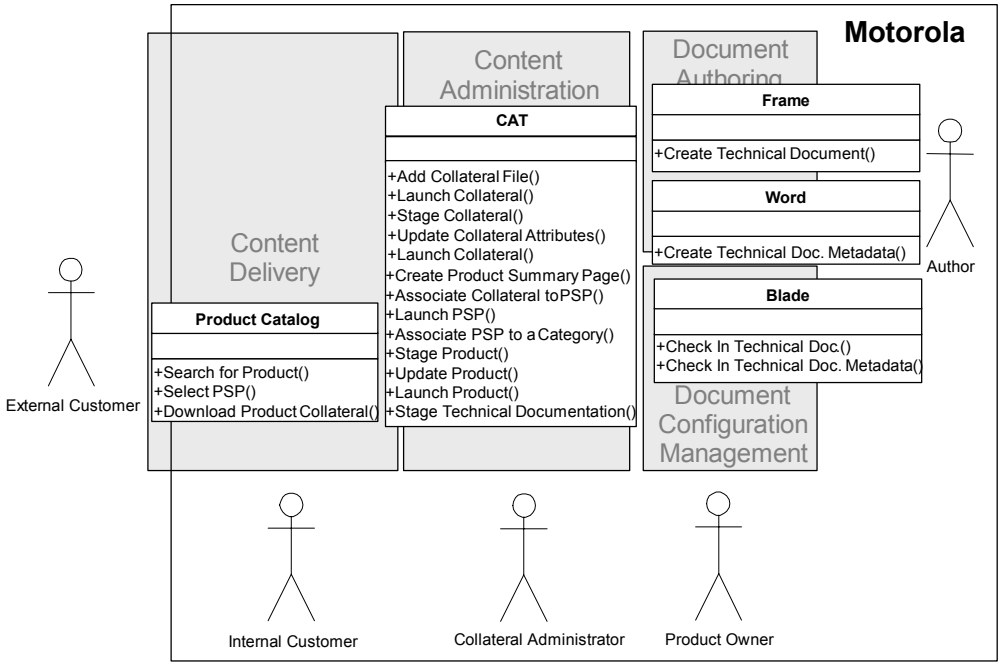


Figure 44 - the eDesign Application Architecture

### 4.4.1 AA Performance Evaluation

The SEPA Application Architecture (AA) encompasses performance characteristics inherent in the application functional and non-functional requirements by specifying the actual allocation of domain functionality amongst TS instances. The choice of which TS instances to employ depends upon satisfying application requirements as well as domain requirements. If more than one TS is available to supply domain functionality, it is useful to provide stakeholders with as much information as possible so they can make an informed TS selection. For this reason, eDesign AA performance evaluations were focused on how the duration of the "Download Product Collateral" service affected its utilization as well as how the service duration affected latency of the "Access Product Information" usage profile. These results can be used to understand how alternative implementations of this critical service might affect overall system performance.

The original "Access Product Information" service duration was specified by eDesign stakeholders in terms of relative duration compared to other services. The "Access Product Information" duration was large: 60 time units compared to 1 time unit for short duration services. Therefore, the AA performance evaluation considered the effects of decreasing the "Download Product Collateral" service duration on: (1) usage profile latency for the "Access Product Information" usage profile, and (2) service utilization for the "Download Product Collateral" service.

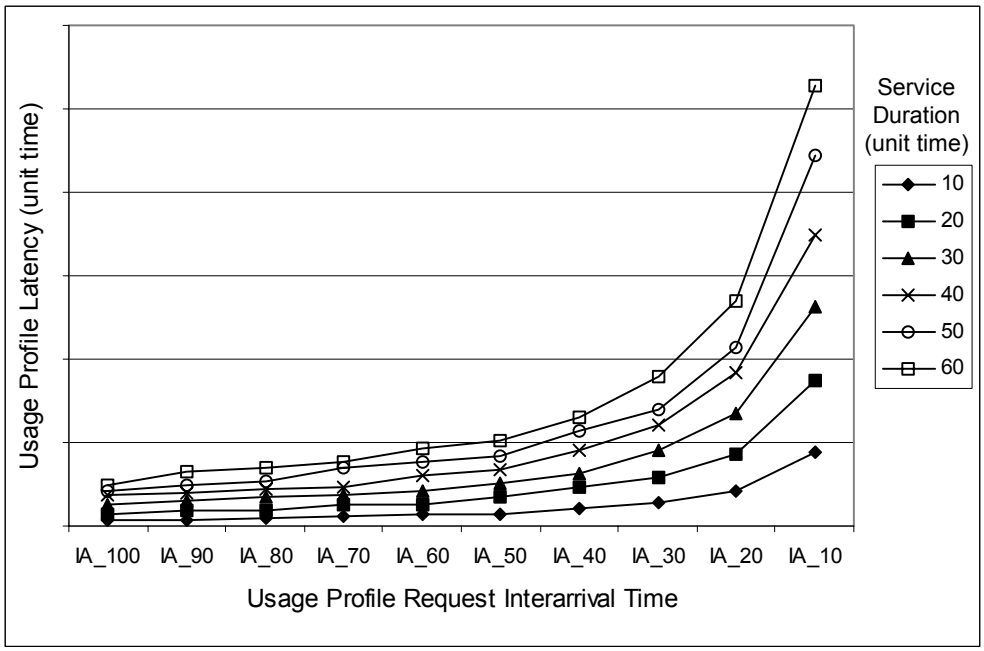


Figure 45 - Effects of Service Duration on Latencies

Figure 46 shows the effects of "Download Product Collateral" service duration on the latency of the "Access Product Information" usage profile. The duration of the service was varied from 10 to 60 simulation time units. Each series in this graph represents the effects of system loading on the usage profile latency when the service duration is set to a specific value. In this figure it can be seen that for each decrease in the duration of "Download Product Collateral" service there is a corresponding improvement in the latency of the "Access Product Information" usage profile.

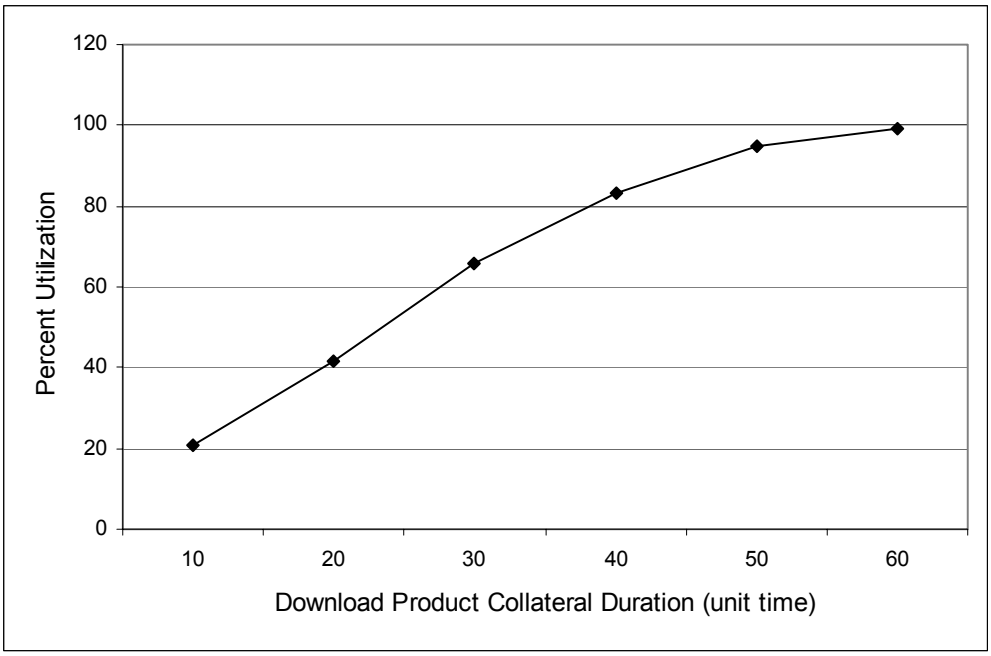


Figure 46 - "Download Product Collateral" Utilization

Figure 46 shows service utilization for the "Download Product Collateral" service as the duration of the service is varied over the range of 10 to 60 simulation time units. Here it can be seen that reducing the duration of the service from 60 to 30 time units or less results in at least a 30% decrease in service utilization.

Results shown in Figure 45 and Figure 46 indicate that a reduction in service duration of 50 percent or greater could have dramatic effects on both usage profile latency and service utilization. While the eDesign team did not have the option to seek an alternative TS instance to provide the "Download Product Collateral" service (the selection of the "Product Catalog" TS to provide this

service was itself driven by an application requirement), these AA performance evaluation results are still useful. For example, from AA performance evaluation it was learned that reductions in the duration of the "Download Product Collateral" service could have a significant positive impact on system performance, therefore it would be useful to consider how IA choices could affect this service duration (either positively or negatively). Lacking the option to use a different software implementation of this functionality, perhaps a hardware allocation strategy in the IA could ensure that performance goals could be met.

### 4.4.2 AA Reliability Evaluation

AA reliability evaluations are useful in understanding how the structural changes introduced by application requirements affect the reliability of the system. In the eDesign case study, there was a considerable amount of reallocation of domain services in the AA as compared to the DRA. One of the bigger changes was the redistribution of the "Author" functionality from a single DRAC in the DRA to three TSs in the AA (e.g., "Word", "Blade", and "CAT"). Therefore, one focus of the eDesign AA reliability evaluation was the effects of these decisions on component reliability ($R_C$).

The graph in Figure 47 shows the reliability sensitivity evaluation results for $R_C$. This graph indicates significant sensitivity of AA reliability ($R_{ARCH}$) to the "CAT", "Word", "Product Catalog", and "Frame" TSs.

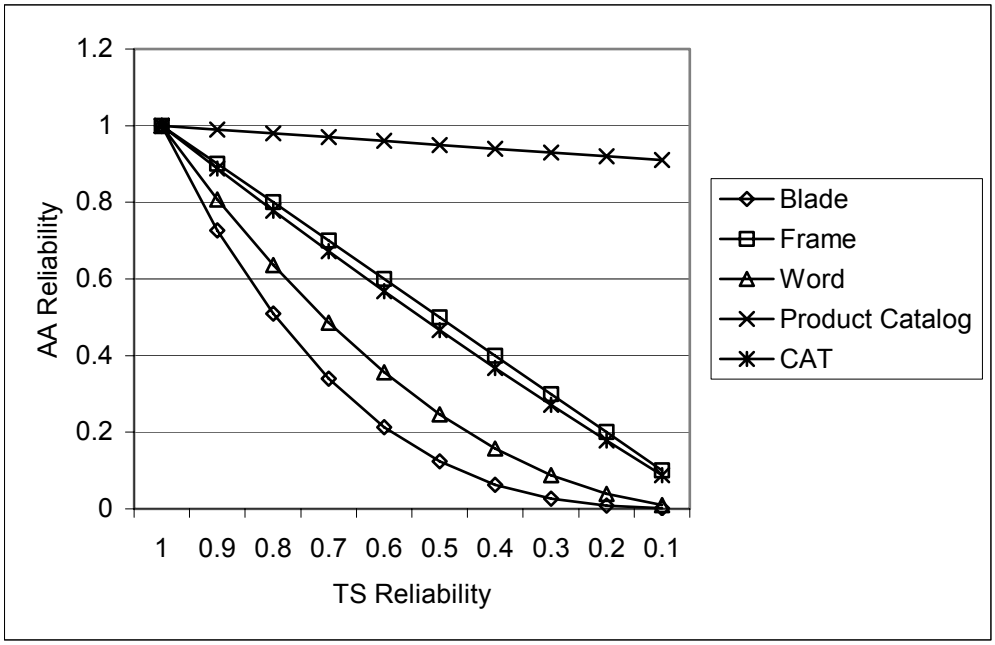


Figure 47 - AA Reliability ($R_C$)

These changes in reliability metrics can be attributed to the re-factoring of highly associated services from few components in the DRA ("Author" and "Publisher"), to more components in the AA ("CAT", "Word", "Product Catalog", and "Frame). This is particularly true of the functionality associated with the "Publish New Technical Document" usage profile. This usage profile occurs with a medium frequency and contains some high and medium duration

services (Table 1). Furthermore, the pre-/post-conditions for the services involved in this usage profile were specified to allow a great deal of flexibility in the order in which services were invoked. For example, while the pre-/post-conditions of the services enforced the execution of "Create Technical Document" prior to "Check In Technical Document", there was no restriction on these services with respect to the order that "Create Technical Document Metadata" and "Check In Technical Document Metadata" were executed (the converse was true as well). As a result, the CDG graph associated with these service executions had many possible traversal paths. Mapping these transitions across multiple components increased the reliability sensitivity by introducing more points of possible failure. The changes in structure are summarized in Table 6.

| SERVICE | ORIGINAL DRAC | TS IN AA |
|---|---|---|
| Create technical document | Author | Frame |
| Create technical document metadata | Author | Word |
| Check in technical document | Author | Blade |
| Check in technical document metadata | Author | Blade |
| Stage technical document | Publisher | CAT |

Table 6- Refactored "Publish New Technical Document"

### 4.4.3 IA Specification and Evaluation

The SEPA Implementation Architecture (IA) supports the satisfaction of site and application installation constraints/requirements, including constraints dictating site-specific hardware platforms, middleware, and communications software (Section 2.2.3). To form an IA, additional TS instances are selected to meet requirements of a specific installation site. The selected TS instances must also satisfy the installation requirements of TSs specified in the AA. Thus, the TS instances in the IA represent the supporting infrastructure on which the AA will be deployed.

The relationship between the AA and the IA is specified in an Installation Map. An Installation Map is comprised of a set of Core Infrastructure Maps and Interaction Maps. The Core Infrastructure Map captures platform information for software deployment (for example, the TS "Product Catalog" requires an instance of a CPU and an Operating System). The Interaction Map captures information relevant to integrating TSs into a larger system (e.g., "CAT" requires an instance of type Middleware to enable communication with "Blade").

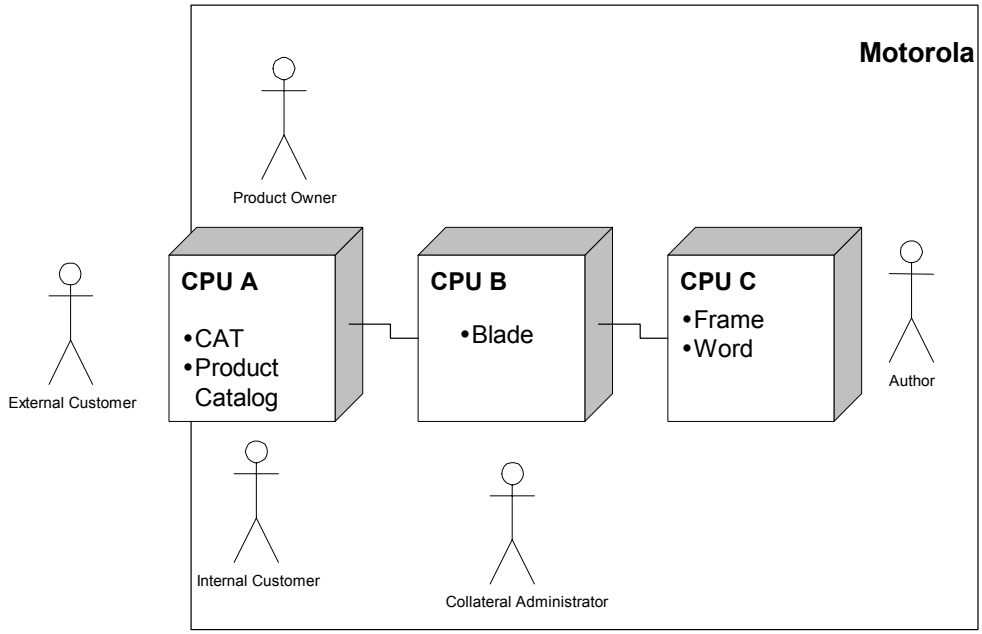


Figure 48 - the eDesign Implementation Architecture

The TS instances in the eDesign AA were allocated among a set of three computing environments to form the eDesign IA, as shown in Figure 48. The stakeholder decisions for mapping the AA to the IA were based upon a desire to co-locate TSs from the AA that had similar functionality. Therefore, "CPU C" supports functionality required by the "Author" role, "CPU B" supports functionality required by the "Collateral Administrator" role, and "CPU C" supports functionality required by the "Product Owner", "Internal Customer", and "External Customer" roles.

### 4.4.4 IA Performance Evaluation

The SEPA Implementation Architecture (IA) encompasses performance characteristics inherent to site and application installation requirements by specifying the relationship of TSs in the AA to additional TSs representing infrastructure components. These infrastructure components have an impact on performance. For example, different CPUs can have different relative execution speeds. Thus, IA performance evaluations can be used to aid stakeholders in determining how their IA choices affect overall system performance.

The eDesign IA performance evaluation continued to focus on the "Access Product Information" usage profile and its associated "Download Product Collateral" service (the "Download Product Collateral" service was previously identified as a performance critical service in DRA and AA performance evaluations in Section 4.3.2 and Section 4.4.1). Because it had been learned during AA performance evaluation that decreasing the duration of the "Download Product Collateral" service could positively affect the latency of the "Access Product Information" usage profile, it was decided to evaluate whether changing the relative execution speed of "CPU A" could improve performance (the "Download Product Collateral" service was allocated to "CPU A"). Therefore, performance evaluations were performed by increasing the relative performance of "CPU A" over a range of one to ten times the speeds of "CPU B" and "CPU C".

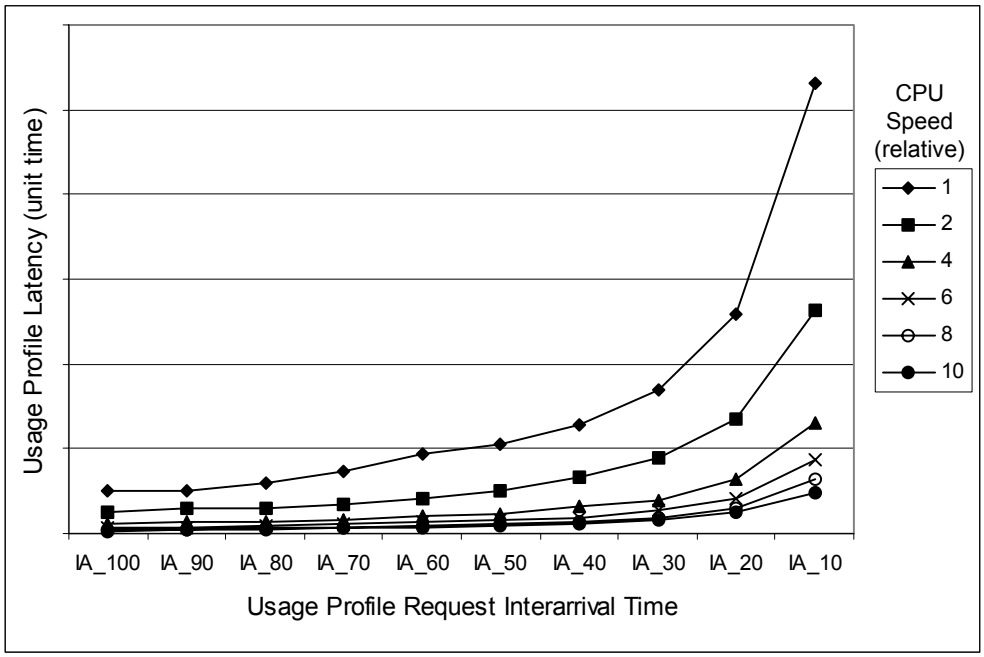


Figure 49 - "Access Product Information" Latencies

Figure 49 shows a graph of the effects of relative CPU speed for "CPU A" on the "Access Product Information" usage profile. This graph shows a range of relative CPU speeds can have a significant effect on the usage profile latency.

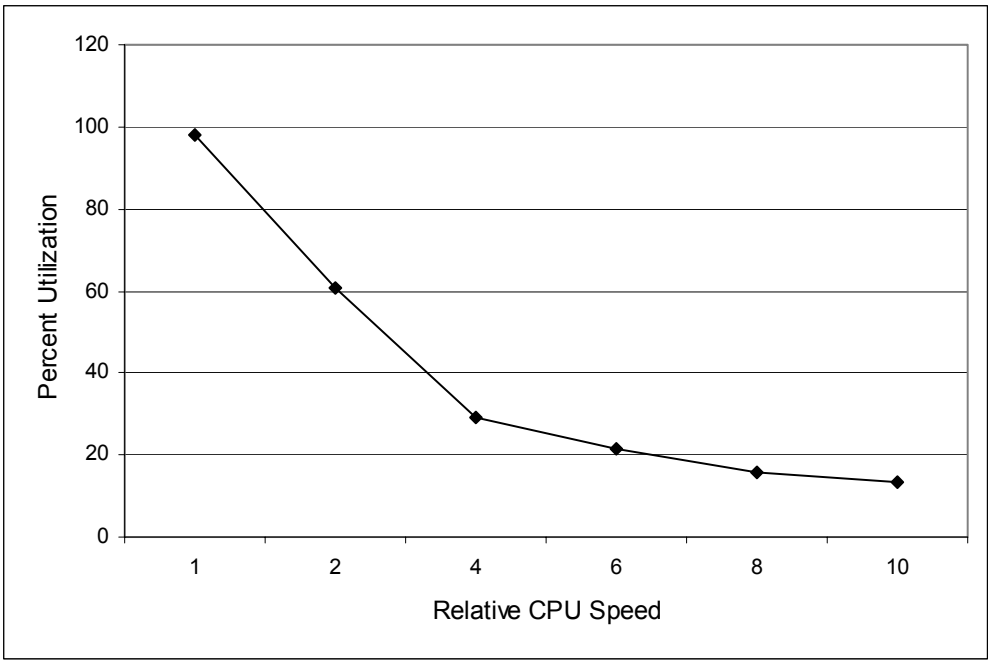


Figure 50 - "Download Product Collateral" Utilization

Figure 50 shows a corresponding graph for service utilization of the "Download Product Collateral" service. This graph illustrates that the relative CPU speed also has a significant impact on the service utilization, which had been identified as a contributing factor to usage profile latency of the "Access Product Information" usage profile (Section 4.3.2 and Section 4.4.1).

Results of the eDesign IA performance evaluation provided valuable information regarding the allocation of TSs to compute environments. These results indicate that the relative execution speed of "CPU A" can potentially have dramatic effects on both "Access Product Information" usage profile latency (Figure 49) and "Download Product Collateral" service utilization (Figure 50).

Furthermore, the relative "CPU A" execution speed increase required to achieve an improvement in usage profile latency and service utilization is economically feasible (e.g., less than an order of magnitude increase in relative execution speed is required for a significant performance improvement). The eDesign stakeholders appreciated this insight and used the information to help understand their compute environment needs.

### 4.4.5 IA Reliability Evaluation

The IA specification determines how TSs from the AA are mapped onto compute environments (Section 2.2.3). Therefore IA reliability evaluations are concerned with understanding how these types of decisions affect reliability of the system.

The eDesign AA reliability evaluation highlighted the "Frame", "Word", and "Blade" TSs as particularly reliability critical due to re-factoring of domain functionality from the DRA amongst TSs (Section 4.4.2). As shown in Figure 48, these TSs were mapped to compute environments "CPU B" and "CPU C". Therefore, of particular interest in the eDesign IA reliability evaluation was the effect of $R_{CE}$ for these compute environments on the overall IA reliability ($R_{ARCH}$).

The graph in Figure 51 shows the IA reliability results for $R_{CE}$. Here it can be seen that $R_{ARCH}$ is most sensitive to the reliability of "CPU C" and "CPU B". This result re-confirmed the reliability evaluation performed for the AA. As a result of the IA reliability evaluation, eDesign developers are reconsidering whether the flexibility of service execution order allowed by the "Publish New Technical Document" domain usage profile (as described in Section 4.4.2) is reasonable in light of the potential reliability issues associated with that flexibility.

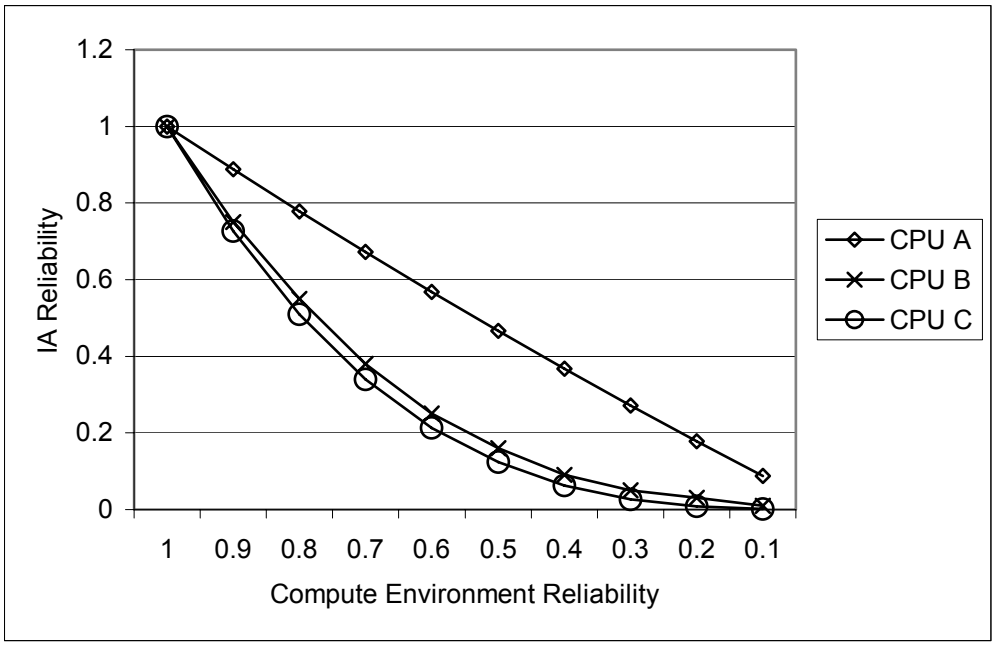


Figure 51 - IA Reliability Evaluation

When the eDesign IA reliability evaluation results are considered in conjunction with IA performance evaluation results some conclusions can be

drawn with regard to critical factors of the eDesign execution environment. For example, "CPU A" is the most performance critical compute environment (because of its association with the "Access Product Information" usage profile - Section 4.4.4), while "CPU B" and "CPU C" are the most reliability critical compute environments. Therefore, when deploying the eDesign system, emphasis should be placed on performance for "CPU A", and reliability for "CPU B" and "CPU C". Furthermore, the communication channel between "CPU B" and "CPU C" should receive attention for reliability because of the high coupling between services in these two environments.

## 4.5 SUMMARY & CONCLUSIONS

The eDesign case study illustrates how Arcade effectively uses the SEPA 3D Architecture to help manage complexity, to reduce the level of expertise required to perform dynamic property evaluations, and to support an iterative approach allowing early, incremental evaluation using partial models. While a small initial investment in time and training was required to adopt the Arcade approach, the eDesign team was subsequently able to gain valuable insights from correctness, performance, and reliability evaluations. These insights were useful in verifying requirements specified in the architecture and in providing rationale for subsequent design and implementation decisions that would have otherwise been supported solely by intuition.

Arcade's incremental approach was effective at mitigating complexity while discovering correctness errors in the eDesign DRA. Correctness evaluation

provided the opportunity to resolve functional errors (completeness, safety, and liveness) before significant architecture commitments had been made. Project participants acknowledged the cost savings in correcting errors in the DRA specification rather than correcting errors detected after system implementation.

From a performance and reliability perspective, Arcade's DRA evaluation yielded useful information early in the development process, highlighting critical architecture elements. These results influenced DRA refinement as well as subsequent design decisions involving application implementation and computing platform selection. For example, having determined the high utilization of the service "Download Product Collateral" during DRA performance evaluations, subsequent eDesign performance evaluations focused on how non-functional and installation requirements affected this critical service. Similarly, the eDesign IA specified an installation requirement to locate Content Delivery functionality (including the "Download Product Collateral" service) on a separate CPU from Document Authoring functionality. The resulting IA performance evaluations suggested that Content Delivery functionality would benefit from a CPU with approximately 4 times the execution speed of the CPU supporting Document Authoring functionality.

A general observation of the eDesign case study is the value of early evaluation. Despite the DRA being a partitioned requirements representation focusing on domain functionality and data, early DRA evaluation results impacted subsequent architecting decisions. Under previously applied evaluation approaches, such errors were not caught until more detailed models (or the actual

implementation) could be constructed (i.e., after significant design decisions were already made).

Results of the eDesign study indicate that the types of evaluations supported by Arcade can yield useful information to system stakeholders early in the systems engineering process. Specifically, the research described herein shows how Arcade can be used to evaluate a DRA to determine dynamic property characteristics inherent to the domain (e.g., the exchange of data/events between business processes, timing between functions, duration of tasks, ordering of tasks as captured in domain usage profiles, and anticipated frequency of execution of domain usage profiles) as well as dynamic property characteristics resulting from possible redistribution of services to TSs in the AA, and eventually the effects of allocating functionality to computing environments in the IA. From these evaluations, the development team can discover important thresholds/boundaries of system behavior governing performance and reliability of the system design and final product.

Application of the SEPA process and Arcade evaluations with the Motorola eDesign project has provided demonstrable benefits to Motorola. One benefit has been to illustrate the value of evaluating dynamic properties early and iteratively during requirements analysis. These evaluations can point out discrepancies between stakeholders' understanding of the domain and analysts' interpretation of the domain knowledge that has been acquired from the stakeholders during requirements analysis, and can also provide rationale for decisions. A second benefit has been in helping Motorola developers recognize

that different types of requirements evolve at different rates, and that partitioning their evaluations along requirements types boundaries allows them a greater opportunity to reuse the knowledge gained from evaluations. In the time since this case study was conducted, the eDesign team has been asked to revise their AA (and consequently their IA) to align with new Motorola corporate standards for certain off-the-shelf TSs (in particular the Blade application) The eDesign developers now realize that their DRA evaluation efforts are still valid with these AA changes, and furthermore the focus on certain critical elements of domain functionality is still valid in evaluating their new AA and IA.

# CHAPTER 5 EXPERIMENTATION

This chapter describes experimentation in support of the hypothesis and research questions defined in Section 1.1. Section 5.1 presents the base data that was used for the experiments. Individual experiments employed and extended this data in various ways. Sections 5.2 through 5.4 describe the general dynamic property evaluation techniques used for experimentation. The remaining sections present experimental setup and results organized by associated research questions.

## 5.1 EXPERIMENTAL DATA

The base data used for this experimentation originated with the eDesign case study (Chapter CHAPTER 4). The case study resulted in a set of SEPA 3D Architecture artifacts capturing domain requirements associated with six major eDesign usage profiles (Table 1). During initial specification of the eDesign requirements, correctness evaluations were performed using partial Domain Reference Architecture (DRA) versions scoped using functional boundaries associated with a single eDesign usage profile (Table 7).

| USAGE PROFILE NUMBER | DOMAIN FUNCTIONALITY |
|---|---|
| $UP_1$ | Publish New Collateral |
| $UP_2$ | Publish New Product |
| $UP_3$ | Access Product Information |
| $UP_4$ | Publish New Technical Document |
| $UP_5$ | Update Collateral |
| $UP_6$ | Update Product |

Table 7 - Functional Scope of eDesign Partial DRA Versions

The process for creating and evaluating partial DRA versions involved iterative specification, correctness property evaluation, and revision (Figure 11). Performance and reliability evaluations were deferred until correctness evaluations were complete. Partial DRA versions were merged at each iteration to produce a full DRA version. Six iterations of the process described above were required to produce a full DRA version to stakeholders' satisfaction.

Henceforth, specific partial DRA versions will be identified as:

$DRA_{REVi,UPj}$

where:

*REVi* indicates the specification/revision iteration from {0..6} (0 indicates the initial specification); and

*UPj* indicates the functional scope of the DRA version according to the usage profile numbers in Table 7.

Specific full DRA versions will be identified as:

$DRA_{REVi}$

where:

*REVi* indicates the specification/revision iteration from {0..6} (0 indicates the initial specification).

At least one partial DRA versions from each of the first five specification/revision iterations exhibited correctness errors. Performance specification errors were detected when performance was evaluated for $DRA_{REV5}$ (these were latent performance attribute specification errors). Reliability was not

evaluated until $DRA_{REV6}$, but because reliability is a measured attribute of the architecture with no reliability-specific specification activities, there were no reliability "errors" recorded (e.g., the DRA contains no reliability-specific information, instead the reliability models are derived from structural, behavioral, and performance information as described in Section 3.2.11). The full DRA versions and the number of correctness and performance errors detected for each version are summarized in Table 8.

| DRA VERSION | SAFETY ERRORS | LIVENESS ERRORS | COMPLETENESS ERRORS | PERFORMANCE ERRORS |
|---|---|---|---|---|
| $DRA_{REV0}$ | 1 | 0 | 2 | 8 |
| $DRA_{REV1}$ | 6 | 3 | 2 | 8 |
| $DRA_{REV2}$ | 6 | 3 | 1 | 8 |
| $DRA_{REV3}$ | 5 | 2 | 7 | 8 |
| $DRA_{REV4}$ | 2 | 0 | 0 | 8 |
| $DRA_{REV5}$ | 0 | 0 | 0 | 8 |
| $DRA_{REV6}$ | 0 | 0 | 0 | 0 |

Table 8 - eDesign Errors by full DRA Version

The process used to create the DRA versions in Table 8 involved employing Arcade in conjunction with the Reference Architecture Representation Environment (RARE) [55]. The process was covered in Section 3.1.5. RARE is a research tool designed to facilitate the derivation of DRAs from domain requirements. During derivation, RARE uses a combination of static property metrics and heuristics to determine the allocation of domain requirements to DRA elements (e.g., functions, data/events, timing, constraints). RARE allows specification of multiple weighted quality goals (associated with both static and

dynamic properties such as maintainability, performance, and reliability) to be employed during the derivation process.

For each DRA version in the base data set (both partial and full), RARE allocated functionality to Domain Reference Architecture Classes (DRACs) that were representative of domain performers (e.g., RARE heuristics were not employed to determine which DRACs should be created or how services should be allocated to them; instead DRACs were defined corresponding to domain performers described by stakeholders, and services were assigned to DRACs according to the associations stakeholders identified between services and domain performers). Henceforth, a DRA version that has this type of allocation of functionality to structure is referred to as a *baseline DRA version*. The iterative eDesign specification and evaluation process employed in the eDesign case study above resulted in 49 baseline DRA versions, summarized below:

- Six partial DRA versions for each of the seven specification/revision iterations (a total of 42 DRA versions), as follows:

  $\{\{DRA_{REV0,UP1}..DRA_{REV6,UP1}\}, \{DRA_{REV0,UP2}..DRA_{REV6,UP2}\},$
  $\{DRA_{REV0,UP3}..DRA_{REV6,UP3}\}, \{DRA_{REV0,UP4}..DRA_{REV6,UP4}\},$
  $\{DRA_{REV0,UP5}..DRA_{REV6,UP5}\},\{DRA_{REV0,UP6}..DRA_{REV6,UP6}\}\}$, and

- Seven full DRA versions (one for each of the seven specification/revision iterations), as follows:

  $\{DRA_{REV0} \ldots DRA_{REV6}\}$.

An Application Architecture (AA) version and an Implementation Architecture (IA) version were also specified during the eDesign case study. Together, the eDesign baseline DRA versions, AA version, and IA version formed the base data set for experimentation described in this chapter.

To support the requirements of individual experiments, the base data set was augmented by using RARE to derive several additional DRA versions. Similarly, additional AA and IA versions were created as required. The additional DRA versions were derived using various quality goals associated with dynamic properties, and thus employed various combinations of weighted RARE dynamic property heuristics (rules of thumb for accomplishing quality goals; recall that RARE assists architects with DRA derivation using a heuristics-based approach that relies on static property metrics applied to a set of weighted quality goals such as maintainability, performance, and reliability as described in Section 3.1.5). The resulting DRA versions contain the same domain functionality as the baseline DRA versions, but exhibit a wide variety of structural allocations. The additional AA and IA versions were created by defining alternative allocations of domain functionality to TSs (e.g., refactoring of domain functionality amongst various applications and computing environments), and by varying performance attributes associated with these TSs. Details of these additional DRA versions, AA versions, and IA versions are provided in sections describing experiments in which they were used.

A common set of evaluation techniques and metrics were used in various ways to support the experiments. Before discussing the individual experiments, the following sections describe these evaluation techniques and metrics, referring back to Section 3.2 as appropriate for definitions.

## 5.2 CORRECTNESS PROPERTY EVALUATION

The following sections describe the common correctness property evaluation techniques and metrics used for experimentation. This is followed by sections discussing performance and reliability metrics and techniques.

### 5.2.1 Correctness Property Evaluation Technique

The correctness property evaluation techniques employed in this experimentation were the SPIN model checking techniques described in Section 3.2.3. Individual experiments controlled the Arcade correctness evaluation parameter `MAX_ERRORS` (for the maximum number of correctness errors to be detected) as appropriate for the given experiment.

### 5.2.2 Correctness Property Metrics

The correctness property metrics used for experimentation were $M(C_S)$: safety errors, and $M(C_L)$: liveness errors (as defined in Section 3.2.4.1 and Section 3.2.5.1, respectively). Individual experiments specified settings for $T(C_S)$ and $T(C_L)$ (threshold values, Section 3.2.12). These settings were used

for reporting correctness property exceptions $EXCP(C_S)$ and $EXCP(C_L)$ (Section 3.2.13).

## 5.3 PERFORMANCE PROPERTY EVALUATION

The following sections discuss the common performance property evaluation techniques and metrics used for experimentation.

### 5.3.1 Performance Property Evaluation Techniques

Performance properties were evaluated using the discrete event simulation techniques implemented in Arcade (Section 3.2.10.2). Summary statistics were collected over multiple simulation runs. Sets of simulation runs were performed over a range of simulated system loading from lightly loaded to heavily loaded. The intent of this technique is to characterize the response of the system under different system loadings. Individual experiments used different settings for the following Arcade simulation parameters: {IA_TIME, TRIALS, SIM_TIME} (Section 3.2.7). IA_TIME is the mean interarrival time between events that initiate usage profile executions, sampled from a negative exponential distribution. TRIALS is the number of simulation runs to perform for each discrete value of IA_TIME. SIM_TIME is the number of simulation time units a simulation run is allowed to execute.

Performance evaluations used the performance property metrics and exceptions defined in Section 3.2.14. The performance property metrics are based upon calculating the ratio of the *expected value* of a property to the *measured*

*value* of a property. The performance metrics are used in conjunction with threshold values. These thresholds are used to detect discrepancies between expected values of property metrics and measured values of property metrics. A measured value for a property metric that is outside of an acceptable range defined by its threshold value is considered a *property exception*.

### 5.3.2 Performance Property Metrics

The performance property metrics used for experimentation were $M(L_{UP})$: usage profile latency, $M(TP_{UP})$: usage profile throughput, $M(U_C)$: component utilization, $M(TP_C)$: component throughput, $M(L_{SVC})$: service latency, $M(U_{SVC})$: service utilization, and $M(TP_{SVC})$: service throughput (as defined in Section 3.2.14). The experiments specified threshold settings for: $T(TP_{UP})$, $T(U_C)$, $T(TP_C)$, $T(L_{SVC})$, $T(U_{SVC})$, and $t(TP_{SVC})$. These settings were used for reporting correctness property exceptions: $EXCP(TP_{UP})$, $EXCP(U_C)$, $EXCP(TP_C)$, $EXCP(L_{SVC})$, $EXCP(U_{SVC})$, and $EXCP(TP_{SVC})$.

## 5.4 RELIABILITY PROPERTY EVALUATION

The following sections describe the common reliability property evaluation techniques and metrics used for experimentation.

### 5.4.1 Reliability Property Evaluation Techniques

The reliability property evaluation techniques employed in this experimentation were the Arcade reliability evaluation techniques described in

Section 3.2.11.4. The SBRA algorithm was used to perform reliability sensitivity evaluations as described in Section 3.2.11.5.

### 5.4.2 Reliability Property Metrics

The reliability property metrics used for experimentation were $M(R_S)$: service reliability, $M(R_C)$: component reliability, and $M(R_{CE})$: compute environment reliability (as defined in Section 3.2.15). The experiments specified settings for $T(R_S)$, $T(R_C)$ and $T(R_{CE})$. These settings were used for reporting correctness property exceptions $EXCP(R_S)$, $EXCP(R_C)$ and $EXCP(R_{CE})$.

The following sections describe experiments associated with specific research questions.

## 5.5 RESEARCH QUESTION #1

*Given artifacts generated during the analysis and design phases of a software engineering process, what artifacts are required to support software architecture execution?*

As noted in Chapter 3, the approach to this research question involved no experimentation. The set of artifacts required to perform dynamic property evaluations of software architectures is discussed in detail in Section 2.1.1. The experiments in this chapter use SEPA 3D Architecture artifacts (Section 2.2).

## 5.6 RESEARCH QUESTION #2

*What dynamic properties and property dependencies can be evaluated by executing the specification of software architectures?*

It has been demonstrated in this dissertation that Arcade can evaluate several dynamic properties by executing the specification of a software architecture. Therefore the main focus of experimentation associated with this research question is evaluation of dynamic property dependencies. The experimental approach used to explore dynamic property dependencies consisted of evaluating a number of different DRA versions with differing structural allocations and differing requirements revision levels. These DRA versions were produced using RARE (Section 5.1, Section 3.1.5). For different DRA versions, RARE's derivation goals were weighted in favor of specific dynamic properties. Each of these DRAs was evaluated with respect to a set of eleven dynamic properties. Evaluation results were then analyzed using statistical correlation techniques.

### 5.6.1 Experiment 1 - Dynamic Property Dependencies

The hypothesis of this experiment is:

> *Early software architecture artifacts can support evaluation of dynamic property dependencies using software architecture execution.*

Changes to an architecture that affect one property in a pair of dependent properties will also affect the other property in the pair (either positively or negatively). These types of dependencies can be measured by changing an architecture in some way (e.g., functionality, structural allocation, scope) and looking for statistically significant correlations between measured values of property pairs. Therefore, the approach chosen to investigate this hypothesis was to evaluate a set of different DRA versions derived from a fixed scope of eDesign requirements (selected from various requirements revision levels, and using RARE to create different structural allocations), and then to perform a statistical analysis of correlation to detect property dependencies (Figure 52). The following sections present the experimental setup, experiment data, and analysis of the results. Detailed results are in APPENDIX D.

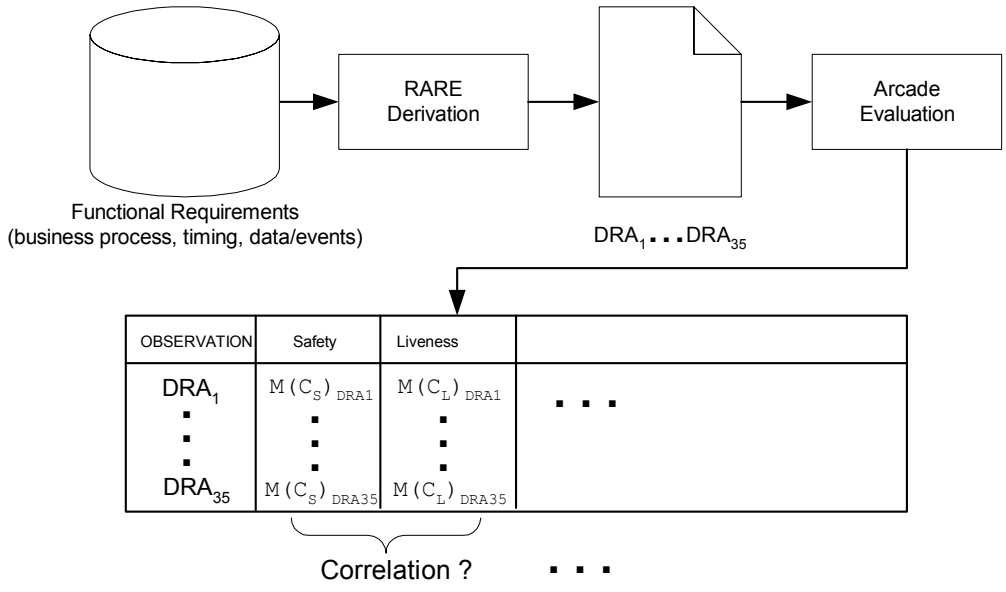


Figure 52 - Experimental Approach for Experiment 1

#### *5.6.1.2 Experiment Setup*

A set of eleven dynamic properties was evaluated for each DRA version in the data set (Table 13). This set of properties is listed in Table 9.

| PROPERTY CLASS | PROPERTY | METRIC |
|---|---|---|
| Correctness | Safety | $M(C_S)$ |
| | Liveness | $M(C_L)$ |
| | | |
| Performance | Usage Profile Latency | $\overline{M(L_{UP})}$ for repeated trials |
| | Usage Profile Throughput | $\overline{M(TP_{UP})}$ for repeated trials |
| | Component (DRAC) Utilization | $\overline{M(U_C)}$ for repeated trials |
| | Component (DRAC) Throughput | $\overline{M(TP_C)}$ for repeated trials |
| | Service Latency | $\overline{M(L_{SVC})}$ for repeated trials |
| | Service Utilization | $\overline{M(U_{SVC})}$ for repeated trials |
| | Service Throughput | $\overline{M(TP_{SVC})}$ for repeated trials |
| | | |
| Reliability | Component (DRAC) Reliability | $\overline{M(R_C)}$ for sensitivity eval. |
| | Service Reliability | $\overline{M(R_{SVC})}$ for sensitivity eval. |

Table 9- Properties Evaluated in Experiment 1

Specific dynamic property evaluation methods and associated Arcade evaluation parameter settings are listed in Table 10. (Arcade's correctness evaluation parameters are described in Section 3.2.3, performance evaluation parameters in Section 3.2.7, and reliability evaluation parameters in Section 3.2.11).

| PROPERTY | METHOD | PARAMETERS |
|---|---|---|
| Correctness | Model Checking | MAX_ERRORS = 10 |
| | | |
| Performance | System Loading | IA_TIME = { 10, 20, 30, 40, 50, 60 , 70, 80 , 90, 100 } |
| | | TRIALS = 10 |
| | | SIM_TIME = 10000 |
| | | |
| Reliability | Sensitivity Eval. | $R_{ELEMp}$ = { 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 } |

Table 10 - Evaluation Methods and Parameters for Experiment 1

Results from all trials were collected and analyzed for statistically significant correlations between all pairs of dynamic property metrics. The statistical analysis tested the following hypotheses for each pair of dynamic property metrics:

$H_0$ - There is no correlation between the pair of dynamic properties

$H_1$ - There is a correlation between the pair of dynamic properties

The evaluation results were first tested for normality using descriptive statistics techniques before selecting an appropriate statistical correlation technique. From these tests, it was determined that the results did not have a normal distribution. This was not unexpected due to the fact that the results were produced by evaluation techniques that are intended to determine how dynamic properties of the system are affected as the system becomes increasingly stressed. This typically produces results with significant positive skew and high kurtosis.

Given that the results were not from a normal distribution, the commonly used Pearson correlation technique was not suitable for this analysis. Therefore,

the Spearman correlation technique was selected [78, 97]. The Spearman correlation is a well-known technique that tests for an association between two related variables. It is the non-parametric alternative to the Pearson correlation, and is equivalent to first ranking the observations, then analyzing the ranks using the Pearson correlation. The Spearman correlation is computed as follows:

$$\rho = 1 - \frac{6\sum D^2}{n(n^2 - 1)}$$

where:

D is the rank difference for an element

N is the number of elements

The Spearman coefficient was calculated for every pair of dynamic properties that were evaluated in this experiment (for a total of 55 pairs). The degrees of freedom were computed as v=33 (where the general formula for degrees of freedom for the Spearman coefficient is v = N-2). The confidence intervals associated with critical values of $\rho$ are summarized in Table 11 [97].

| CORRELATION ? | POSSIBLE | PROBABLE | VERY PROBABLE | ALMOST CERTAIN |
|---|---|---|---|---|
| Confidence Interval | 0.1000 | 0.0500 | 0.0200 | 0.0100 |
| Value of $\rho$ ($\pm$) | 0.2830 | 0.3350 | 0.3940 | 0.4330 |

Table 11 - Critical Values of $\rho$ for Experiment 1

Once a probable correlation was established (to a confidence of 95% or higher), a two-tailed *p-value* was used to judge the significance of the Spearman coefficient. The two-tailed test was selected since this experiment was designed to identify whether any form of correlation (e.g., either positive or negative) existed between dynamic properties under evaluation. When the *p-value* was below a predetermined cut-off point, known as the significance level (formally expressed as the alpha, or $\alpha$-level), it was considered significant. The $\alpha$–level for this test was set to 0.05, therefore a p-value $\leq$ 0.05 was considered to be significant (indicating a 95% confidence in the significance).

For each significant Spearman coefficient, a linear regression was performed to further determine the significance of the correlation. Correlations that exhibited a high degree of confidence and significance, but which did not yield good linear regression results were rejected. For this experiment, the $R^2$ value calculated by the linear regression was used to determine the goodness-of-fit. The $R^2$ statistic indicates how much of the variation within the sample is accounted for by the fitted regression line. Values closer to 1.0 indicate much variation has been accounted for by the regression. Table 12 summarizes the $R^2$ threshold used to judge goodness-of-fit for this experiment.

| GOODNESS-OF-FIT | WEAK | MODERATE | STRONG |
|---|---|---|---|
| $R^2$ Value | $0.5 < R^2 < 0.7$ | $0.7 < R^2 < 0.9$ | $0.9 < R^2$ |

Table 12 - Goodness-of-Fit Values for Experiment 1

#### *5.6.1.3 Experiment Data*

The data for this experiment consisted of a set of DRA versions derived from the full set of requirements represented by eDesign $DRA_{REV6}$,…,$DRA_{REV6}$. These DRA versions were not baseline eDesign DRA versions (Section 5.1), they were derived specifically for this experiment using RARE.

The set of DRA versions used for this experiment is shown in Table 13. Each DRA version resulted from specific dynamic property goal weightings intended to influence RARE's choice of heuristics governing structural allocation to emphasize correctness, performance, or reliability. The degree to which correctness heuristics were employed was governed by (1) the weight of the correctness property goal and (2) the choice of which requirements revision level the associated DRA version would be derived from (where later requirements revision levels were more correct than earlier revisions). In some cases RARE was configured to emphasize multiple dynamic properties, but always with a stronger preference towards only one property. The resulting data set consisted of 35 DRA versions. The particular dynamic property emphasis of each DRA version is summarized in Table 13. In cases where the derivation emphasis is listed as "-" this indicates that RARE gave no special consideration to this property when deriving a DRA structure.

| DRA VERSION | CORRECTNESS EMPHASIS | PERFORMANCE EMPHASIS | RELIABILITY EMPHASIS | OVERALL PROPERTY EMPHASIS |
|---|---|---|---|---|
| 1 | - | 1 | 2 | performance |
| 2 | - | 2 | 1 | reliability |
| 3 | 1 | - | - | correctness |
| 4 | - | - | 1 | reliability |
| 5 | - | 1 | - | performance |
| 6 | - | 1 | 2 | performance |
| 7 | - | 2 | 1 | reliability |
| 8 | 1 | - | - | correctness |
| 9 | - | - | 1 | reliability |
| 10 | - | 1 | - | performance |
| 11 | - | 1 | 2 | performance |
| 12 | - | 2 | 1 | reliability |
| 13 | 1 | - | - | correctness |
| 14 | - | - | 1 | reliability |
| 15 | - | 1 | - | performance |
| 16 | - | 1 | 2 | performance |
| 17 | - | 2 | 1 | reliability |
| 18 | 1 | - | - | correctness |
| 19 | - | - | 1 | reliability |
| 20 | - | 1 | - | performance |
| 21 | - | 1 | 2 | performance |
| 22 | - | 2 | 1 | reliability |
| 23 | 1 | - | - | correctness |
| 24 | - | - | 1 | reliability |
| 25 | - | 1 | - | performance |
| 26 | - | 1 | 2 | performance |
| 27 | - | 2 | 1 | reliability |
| 28 | 1 | - | - | correctness |
| 29 | - | - | 1 | reliability |
| 30 | - | 1 | - | performance |
| 31 | - | 1 | 2 | performance |
| 32 | - | 2 | 1 | reliability |
| 33 | 1 | - | - | correctness |
| 34 | - | - | 1 | reliability |
| 35 | - | 1 | - | performance |

Table 13 - Experimental data for Experiment 1

The heuristics employed by RARE to produce the DRA versions listed in Table 13 are described as follows.

RARE Completeness/Correctness Heuristics

(1) DRAC-related

- Build initial DRACs based on performer roles.
- Correct missing data/event destination/source in service I/O definitions.
- Add DRACs for roles not already covered.
- Define data attributes for concepts not accounted for.

RARE Performance Heuristics

(1) DRAC-related

- Build initial DRACs based on task sequence (i.e., temporal locality).

(2) Subsystem-related[1]

- When high levels of DRAC service dependencies are detected, collect respective DRACs into a subsystem.
- Build subsystems based on highly used resources.
- Build subsystems based on service sequences from usage profiles.

[1] Arcade views RARE subsystems as a single component

RARE Reliability Heuristics

(1) DRAC-related

- Build initial DRACs based on task location (i.e., spatial locality).
- Isolate heavily used attributes.
- Split DRAC if input or output count is too large, suggesting high complexity.

#### *5.6.1.4 Analysis and Conclusions*

Arcade evaluation was able to detect a number of dependencies between dynamic property pairs for the 35 DRA versions in the experimental data set. With strict limits for acceptance of a correlation, there were 8 significant correlations detected between the possible 55 pairings of the 11 properties (e.g., about 15% of the pairs of dynamic properties showed a correlation). Details of these findings can be seen in Table 37 through Table 50 in APPENDIX D.

For a number of property pairs the Spearman coefficient indicated a high confidence and significance for correlation, but the $R^2$ value did not meet the acceptance standards specified in Table 12. In these cases the linear regressions for these pairs of properties indicated too much unexplained variance in the results and the correlation was rejected. In several instances the apparent correlations that were rejected by the linear regression analysis offer evidence of correlations that exist but cannot be explained by simple linear regressions. For example, the linear regression for $TP_{UP}$ and $L_{SVC}$ (usage profile throughput and service latency) is shown in Figure 53. This regression had an $R^2$ value of 0.46

(below the 0.5 acceptance level for $R^2$) , while the Spearman correlation indicated a 99% confidence in rejecting $H_0$ (e.g., in this case there is strong statistical evidence of a correlation, but the variance is not fully explained by a simple linear regression using the two dynamic properties).

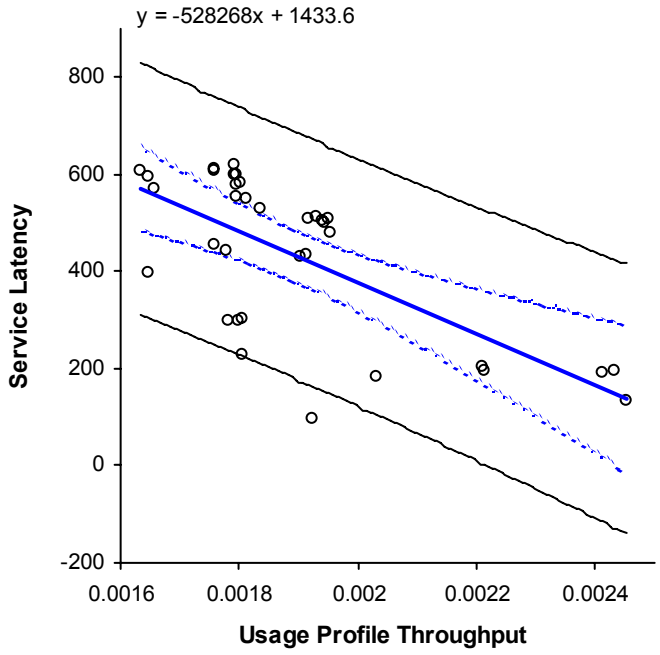


Figure 53 - Linear Regression of $TP_{UP}$ and $L_{SVC}$

Aggregating the correlation results into various comparison groups provides some additional insights. For example, correlations between the correctness, performance, and reliability property classes are summarized in Table 14. This table lists the percentage of properties within and across property classes that had correlations. Correlations between properties within the correctness class

were 100%, indicating that safety and liveness were strongly correlated with each other. Correctness properties did not show any correlation to performance properties, and 50% of correctness properties showed a correlation with at least one reliability property. Approximately 20% of the performance properties showed a correlation with at least one other performance property (although many property pairs hinted at correlations as mentioned above), but no correlations were detected between performance and reliability. Reliability properties (e.g., component reliability and service reliability) did not indicate a correlation between each other.

| | CORRECTNESS | PERFORMANCE | RELIABILITY |
|---|---|---|---|
| CORRECTNESS | 100.00% | | |
| PERFORMANCE | 0.00% | 20.41% | |
| RELIABILITY | 50.00% | 0.00% | 0.00% |

Table 14- Summary of Correlations Between Property Classes

Table 15 summarizes results with the property classes divided into groups associated with usage profiles, components, and services. This breakdown shows that the correlation between correctness properties and reliability properties is only associated with service reliability. Furthermore, the only performance property group that has a correlation to other performance property groups is the usage profile performance property group.

| | CORRECTNESS | USAGE PROFILE PERFORMANCE | COMPONENT PERFORMANCE | SERVICE PERFORMANCE | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|
| CORRECTNESS | 100.00% | | | | | |
| USAGE PROFILE PERFORMANCE | 0.00% | 0.00% | | | | |
| COMPONENT PERFORMANCE | 0.00% | 25.00% | 50.00% | | | |
| SERVICE PERFORMANCE | 0.00% | 33.33% | 0.00% | 22.22% | | |
| COMPONENT RELIABILITY | 0.00% | 0.00% | 0.00% | 0.00% | 100.00% | |
| SERVICE RELIABILITY | 100.00% | 0.00% | 0.00% | 0.00% | 0.00% | 100.00% |

Table 15 - Correlations by Usage Profile, Component, and Service

The lack of a correlation between performance properties and correctness and reliability properties can be attributed to the way in which the various evaluation models are constructed by Arcade prior to evaluation by the architecture execution tools. Arcade's performance model abstracts away correctness details such as pre-/post-condition logic associated with the exchange of data and events. Thus, the conditions that give rise to safety and liveness errors are not modeled during performance evaluations; they are assumed to have been checked by the model checker. Similarly, the Arcade performance evaluation method does not consider reliability attributes of the models, nor does it model failures of architecture elements. Again, these reliability attributes are assumed to have been evaluated using other Arcade techniques.

The presence of a correlation between correctness and reliability properties can also be attributed to the way in which models are constructed for the evaluation tools. The edges in the graph models used for the SBRA reliability calculation are constructed by examining the Execution Space generated by

Arcade correctness evaluations (Section 3.2.6.3). Therefore, correctness errors affect the CDG model used for reliability evaluation. This in turn affects the overall reliability calculation performed by the SBRA algorithm.

These experimental results affirm the hypothesis stated in Section 5.6.1 by demonstrating that the information contained in early software architecture models is sufficient to support detection of dynamic property dependencies.

## 5.7 RESEARCH QUESTION #3

*What is the systematic process and related techniques necessary to execute and evaluate a software architecture specification?*

As noted in Chapter 3, the approach to this research question involved no experimentation. See Section 2.1.3 for a summary of findings associated with this research question.

## 5.8 RESEARCH QUESTION #4

*Given decisions during the analysis and design phases of a software engineering process, which decisions can prove to impact the dynamic properties under software architecture execution evaluations?*

The experiments associated with this research question were designed to examine two classes of decisions: (1) decisions associated with the allocation of functionality to structure, and (2) decisions associated with requirements

evolution. The technique of measuring dynamic property exceptions to produce architecture rankings is used for the experimentation. The definitions of dynamic property exceptions were presented in Section 3.2.13, Section 3.2.14, and Section 3.2.15. The ranking technique is described in Section 3.2.16.

### 5.8.1 Experiment 2 - Structural Decisions

The hypothesis of this experiment is:

*Early software architecture artifacts can support evaluation of the effects of structural decisions on dynamic properties.*

Structural decisions occur during the process of allocating functionality to structure. The general approach selected for this experiment was to produce a set of DRA versions derived from a common set of eDesign requirements, and to rank those DRA versions using the Arcade architecture comparison techniques described in Section 3.2.16. The various DRA versions differed only in their allocation of functionality to structure (e.g., there were different allocations of services amongst DRACs, but all services were allocated to a DRAC). Since the DRA versions are derived from the same set of requirements at the same requirements revision level (e.g., `REV6`), the DRA version rankings can be attributed to the effects of structural decisions. The overall approach for one dynamic property is depicted in Figure 54.

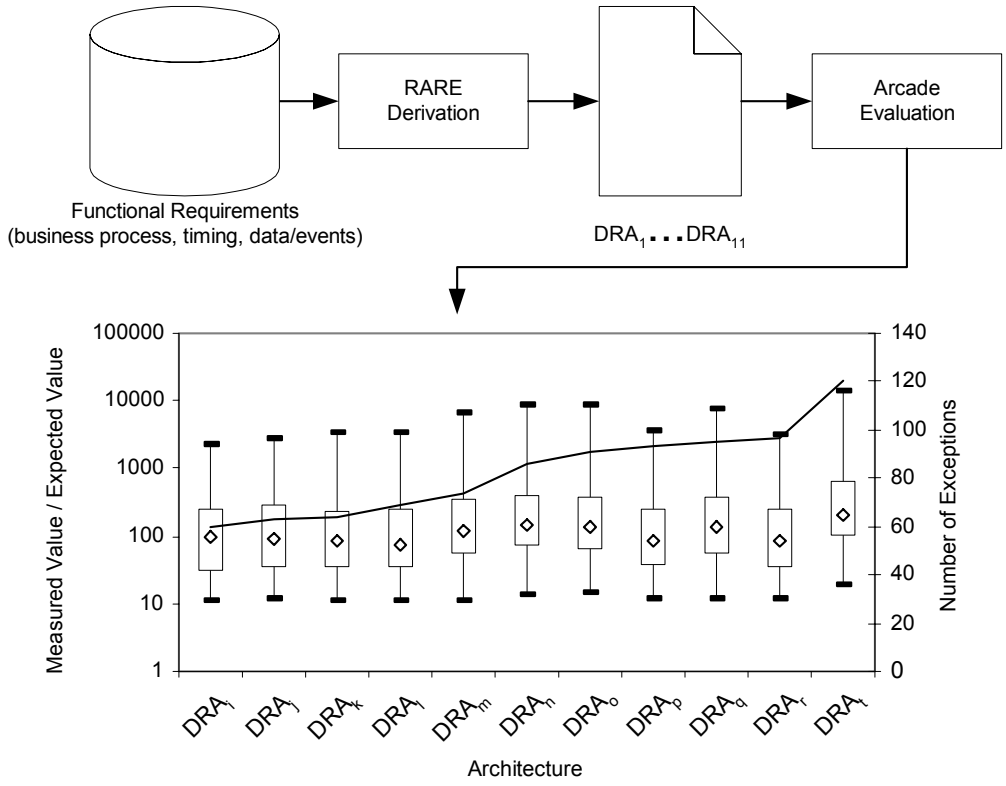


Figure 54 - Experimental Approach for Experiment 2

APPENDIX D presents the results for each property evaluated in this experiment using box-and-whiskers plots (representing the descriptive statistics for property exceptions) overlaid with a line graph representing the DRA version ranking based upon the number of property exceptions. An example of this is shown in Figure 54. The box-and-whiskers portion of the charts are associated with the left-hand Y-axis. The top-most and bottom-most hash marks on each box represent min and max. The box is bounded on the lower side by the 1st quartile statistic and on the upper side by the 3rd quartile statistic. The diamond shape marks the median for property exceptions. The line graph represents the

number of property exceptions, and is associated with the right-hand Y-axis. Results for all DRA versions are shown in ascending order left to right by number of property exceptions. This ordering is the rank ordering assigned to the DRA versions for a given property. Rankings are calculated in this way for each property evaluated. The following sections present the experimental setup and data, followed by analysis of the results.

#### *5.8.1.2 Experiment Setup*

Evaluations were limited to a set of nine performance and reliability properties for this experiment since this is sufficient to establish that structural decisions affect multiple properties and property classes. This set of properties is listed in Table 16. Performance property metrics and exceptions are described in Section 3.2.14, reliability property metrics and exceptions are described in Section 3.2.15.

| PROPERTY CLASS | PROPERTY | METRICS |
|---|---|---|
| Performance | Usage Profile Latency | $EXCP(L_{UP})$ |
| | Usage Profile Throughput | $EXCP(TP_{UP})$ |
| | Component (DRAC) Utilization | $EXCP(U_C)$ |
| | Component (DRAC) Throughput | $EXCP(TP_C)$ |
| | Service Latency | $EXCP(L_{SVC})$ |
| | Service Utilization | $EXCP(U_{SVC})$ |
| | Service Throughput | $EXCP(TP_{SVC})$ |
| | | |
| Reliability | Component (DRAC) Reliability | $EXCP(R_C)$ |
| | Service Reliability | $EXCP(R_{SVC})$ |

Table 16- Properties Evaluated in Experiment 2

The evaluation methods and associated Arcade evaluation parameters used for evaluating this set of properties are listed in Table 17. (Arcade's performance evaluation parameters are discussed in Section 3.2.7, and reliability evaluation parameters in Section 3.2.11).

| PROPERTY | METHOD | PARAMETERS |
|---|---|---|
| Performance | System Loading | IA_TIME = { 10, 20, 30, 40, 50, 60 , 70, 80 , 90, 100 } |
| | | TRIALS = 10 |
| | | *SIM_TIME* = 10000 |
| | | |
| Reliability | Sensitivity Eval. | $R_{ELEMp}$ = { 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 } |

Table 17 - Evaluation Methods and Parameters for Experiment 2

The exception threshold values for each of the property metrics are listed in Table 18 (for definitions see Section 3.2.14 and Section 3.2.15). Property exception statistics were calculated for each set of property exceptions measured for each DRA version over repeated trials. The experimental results also report descriptive statistics for the metrics shown in Table 16 including: median, min, max, 1st quartile, and 3rd quartile. These descriptive statistics are reported for property metric values identified as exceptions.

| PROPERTY CLASS | PROPERTY | THRESHOLD |
|---|---|---|
| Performance | Usage Profile Latency | $T(L_{UP})$ = 10.0 |
| | Usage Profile Throughput | $T(TP_{UP})$ = 0.33 |
| | Component (DRAC) Utilization | $T(U_C)$ = 0.33 |
| | Component (DRAC) Throughput | $T(T_C)$ = 0.33 |
| | Service Latency | $T(L_{SVC})$ = 10.0 |
| | Service Utilization | $T(U_{SVC})$ = 0.33 |
| | Service Throughput | $T(TP_{SVC})$ = 0.33 |
| | | |
| Reliability | Component (DRAC) Reliability | $T(R_C)$ = 0.25 |
| | Service Reliability | $T(R_{SVC})$ = 0.25 |

Table 18 - Threshold Settings for Property Exceptions

#### *5.8.1.3 Experiment Data*

The data for this experiment consisted of a set of DRA versions derived from the full set of requirements represented by eDesign $\mathrm{DRA}_{\mathrm{REV6}}$. These DRA versions were not baseline DRA versions (Section 5.1), they were derived from eDesign requirements specifically for this experiment using RARE. Table 19 summarizes these DRA versions.

| DRA VERSION | DRA NAME | ALLOCATION OF USAGE PROFILES TO DRACS (TABLE 7) |
|---|---|---|
| 1 | V01_P1-6 | DRAC1: $UP_1..UP_6$ |
| | | |
| 2 | V02_P123P456 | DRAC1: $UP_1$, $UP_2$, $UP_3$ |
| | | DRAC2: $UP_4$, $UP_5$, $UP_6$ |
| | | |
| 3 | V03_P234P561 | DRAC1: $UP_2$, $UP_3$, $UP_4$ |
| | | DRAC2: $UP_5$, $UP_6$, $UP_1$ |
| | | |
| 4 | V04_P345P612 | DRAC1: $UP_3$, $UP_4$, $UP_5$ |
| | | DRAC2: $UP_6$, $UP_1$, $UP_2$ |
| | | |
| 5 | V05_P135P246 | DRAC1: $UP_1$, $UP_3$, $UP_5$ |
| | | DRAC2: $UP_2$, $UP_4$, $UP_6$ |
| | | |
| 6 | V06_P256P134 | DRAC1: $UP_2$, $UP_5$, $UP_6$ |
| | | DRAC2: $UP_1$, $UP_3$, $UP_4$ |
| | | |
| 7 | V07_P12P34P56 | DRAC1: $UP_1$, $UP_2$ |
| | | DRAC2: $UP_3$, $UP_4$ |
| | | DRAC3: $UP_5$, $UP_6$ |
| | | |
| 8 | V08_P14P36P52 | DRAC1: $UP_1$, $UP_4$ |
| | | DRAC2: $UP_3$, $UP_6$ |
| | | DRAC3: $UP_5$, $UP_2$ |
| | | |
| 9 | V09_P13P45P26 | DRAC1: $UP_1$, $UP_3$ |
| | | DRAC2: $UP_4$, $UP_5$ |
| | | DRAC3: $UP_2$, $UP_6$ |
| | | |
| 10 | V10_P16P32P54 | DRAC1: $UP_1$, $UP_6$ |
| | | DRAC2: $UP_3$, $UP_2$ |
| | | DRAC3: $UP_5$, $UP_4$ |
| | | |
| 11 | V11_P15P24P36 | DRAC1: $UP_1$, $UP_5$ |
| | | DRAC2: $UP_2$, $UP_4$ |
| | | DRAC3: $UP_3$, $UP_6$ |

Table 19 - DRA Versions Used for Structural Decision Experiments

The DRA versions shown in Table 19 included:

1. A DRA version consisting of an individual DRAC containing services supporting all eDesign usage profiles (Table 7),
2. A set of DRA versions consisting of all possible combinations of two DRACs, where each DRAC contained services required to support half of the eDesign usage profiles, and
3. A set of DRA versions consisting of all possible combinations of three DRACs, where each DRAC contained services required to support one third of the eDesign usage profiles.

#### *5.8.1.4 Analysis and Conclusions*

The relative rankings of DRA versions according to property exceptions takes on a well-defined pattern with respect to favoring certain architectural structures. This pattern is shown in Table 20. In this table, the entry associated with each kind of property exception summarizes how the property exception metrics ranked the DRA versions with respect to the number of DRACs in the DRA (DRA versions with fewer property exceptions are ranked higher than DRA versions with more exceptions). Exception metrics based on components (e.g., $EXCP(U_C)$: component utilization, $EXCP(TP_C)$: component throughput, and

$EXCP(R_C)$ : component reliability) consistently ranked DRA versions with fewer DRACs higher. Here the entry "1>2>3" indicates that DRA versions with 1 DRAC were ranked higher than DRA versions with 2 DRACs which in turn were ranked higher than DRA versions with 3 DRACs. Conversely, exception metrics based on usage profiles and services (e.g., $EXCP(L_{UP})$ : usage profile latency, $EXCP(TP_{UP})$ : usage profile throughput, $EXCP(L_{SVC})$ : service latency, $EXCP(U_{SVC})$ : service utilization, and $EXCP(TP_{SVC})$ : service throughput) consistently ranked DRA versions with more DRACs higher than DRA versions with fewer DRACs. The notable exception to this pattern is $EXCP(R_S)$ - service reliability. There was not a discernable pattern for this metric.

| $EXCP(L_{UP})$ | $EXCP(TP_{UP})$ | $EXCP(U_C)$ | $EXCP(TP_C)$ | $EXCP(L_{SVC})$ | $EXCP(U_{SVC})$ | $EXCP(TP_{SVC})$ | $EXCP(R_C)$ | $EXCP(R_S)$ |
|---|---|---|---|---|---|---|---|---|
| 3>2>1 | 3>2>1 | 1>2>3 | 1>2>3 | 3>2>1 | 3>2>1 | 3>2>1 | 1>2>3 | UNKNOWN |

Table 20 - Ranking Patterns for $EXCP$(*) (Number of DRACs)

These patterns can be explained by observing that component metrics such as utilization and throughput are optimized when services are distributed across fewer components, whereas metrics associated with services and usage profiles are optimized when execution is distributed across more components.

The graph in Figure 55 shows the rankings of the DRA versions as determined by averaging the rankings for all performance property exceptions (e.g., for a given DRA version, the rankings for each individual performance property are summed and then divided by the number of performance properties). The graph in Figure 56 shows similar information for reliability properties.

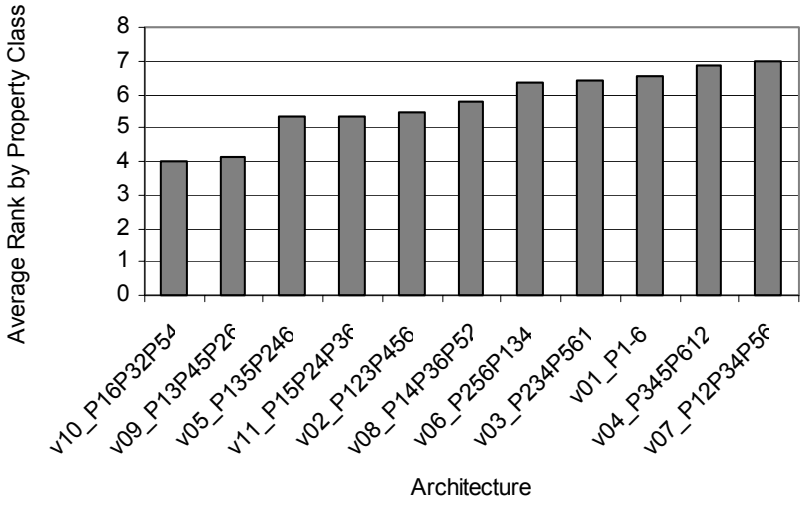


Figure 55 - Average Performance Property Ranking

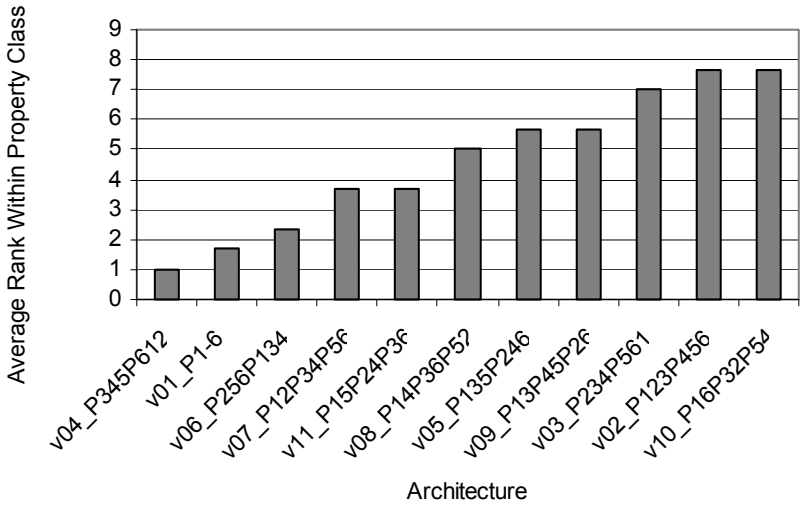


Figure 56 - Average Reliability Property Ranking

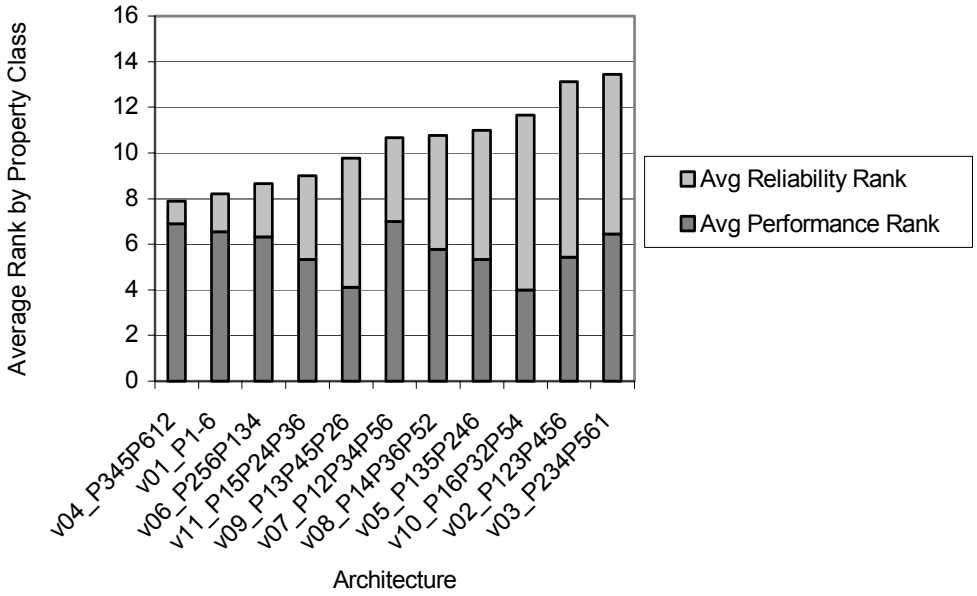


Figure 57 - Cumulative DRA Version Rankings

The graph in Figure 57 shows how the average rankings for performance and reliability properties can be stacked to represent an overall ranking of the DRA versions (using the technique described in Section 3.2.16). Derivation of the DRA versions in this experiment involved decisions about how many DRACs to create, and how many services to allocate to each DRAC. The results of those decisions can be seen in a clear way in Figure 57. There is no absolute best DRA version, but it can be seen from the graphs in Figure 55, Figure 56, and Figure 57 that DRA version v04_P235P612 had the fewest reliability exceptions (but was not the best performer), DRA version v10_P16P32P54 had one of the best performance rankings (but was not the best at reliability), and perhaps DRA

version v11_P15P24P38 or DRA version v09_P13P45P28 provide the best compromise for both performance and reliability.

In summary, there are clear and explainable patterns of rankings of the DRA versions used in this experiment. As demonstrated by the discussion above, these results can be explained in terms of the structural decisions associated with the ranked DRA versions. Therefore, the findings of this experiment support the hypothesis that the effects of structural decisions can be evaluated using early software architecture artifacts.

### 5.8.2 Evolution Decisions Experimentation

Requirements evolution occurs as an initial set of requirements matures, or as a set of requirements changes over time. As evolution occurs, stakeholders must make decisions with respect to various aspects of their architectures. While it is not possible to examine all types of these decisions, the intent of this experimentation is to examine the effects of decisions related to requirements evolution along two dimensions supported by the SEPA 3D Architecture:

1. *Within-Architecture Decisions*: these decisions are made as requirements evolve with respect to a single architecture (for example a DRA), and
2. *Across-Architecture Decisions*: these decisions are made as requirements evolve with respect to two related architectures (for example a DRA and an AA).

Therefore, two separate experiments were performed, one to examine within-architecture evolution, and one to examine across-architecture evolution. These experiments are presented in Section 5.8.3 and Section 5.8.4 respectively.

### 5.8.3 Experiment 3 - Within-Architecture Evolution

The hypothesis for this experiment is:

*Dynamic properties of early software architectures qualitatively change as a function of evolution within the architecture.*

This experiment examines the effects of two types of within-architecture decisions on the dynamic properties of an architecture. The first type of within-architecture decisions are *scoping decisions* whereby the scope of functionality included in an architecture is determined. In the eDesign case study, the scope of a DRA version was determined by the usage profiles included in that DRA version (e.g., there were different partial DRA versions and full DRA versions whose scopes were determined by the usage profiles in Table 7, see Section 4.3). The second type of within-architecture decisions are *correction/refinement decisions* whereby the details of functional and non-functional requirements in an architecture are adjusted. Examples of these types of decisions in the eDesign case study were the choices of how to address safety or liveness errors (e.g., adjusting pre-/post-conditions or inputs/outputs), and the decision to spread domain functionality originally associated with a single service over multiple

services (Section 4.3.1). Given these two types of within-architecture decisions, the overall approach of this experiment is illustrated in Figure 58.

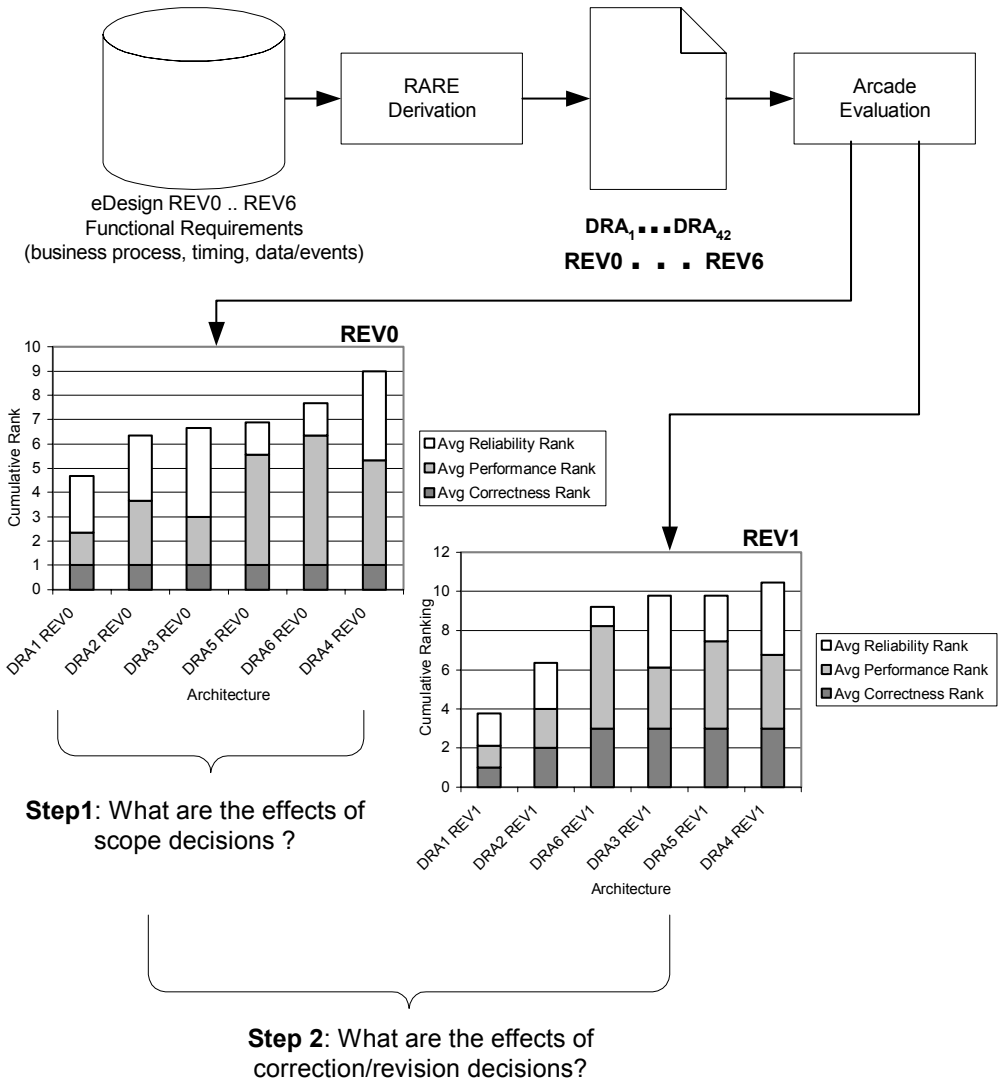


Figure 58 - Experimental Approach for Experiment 3

The process depicted in Figure 58 begins with the set of functional requirements associated with all eDesign usage profiles listed in Table 7. In the eDesign case study, these requirements underwent several revisions during the course of correction/refinement while the overall scope of requirements remained relatively stable (e.g., the same six usage profiles existed for all requirements revision levels). To vary the scope of requirements for this experiment, a set of six DRA versions were derived for *each* requirements revision level. The scope of a DRA version in this set was controlled by the number of usage profiles that were included. For example DRA1 REV0 included the functionality for usage profile 1, at requirements revision level 0, while DRA2 REV0 included the functionality for both usage profile 1 and usage profile 2, at requirements revision level 0. Similarly DRA1 REV1 included the functionality for usage profile 1, at requirements revision level 1 (see Table 27 for more details on DRA versions used in this experiment). Figure 59 shows the set of DRA versions that were derived for each eDesign requirements revision level.

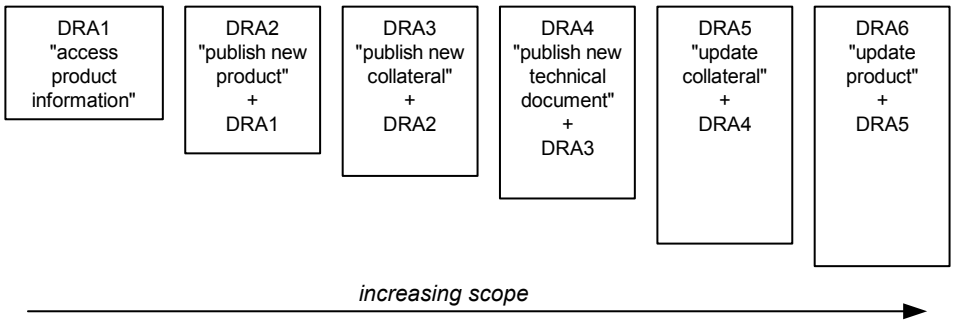


Figure 59 - DRA Scopes for Each Requirements Revision Level

For each requirements revision level, the set of six DRA versions were evaluated and ranked to determine if the effects of scoping decisions could be detected (step (1) in Figure 58). Following that, an analysis of correlation across revision levels was performed to determine if the scoping decisions had a greater impact on the rankings of DRA versions than the correction/refinement decisions (step (2) in Figure 58). If the rankings of the six DRA versions relative to each other shows a high correlation across requirements revision levels (e.g. the six DRA versions were ranked similarly regardless of requirements revision level), then the effects of scoping decisions were greater than the effects of correction/refinement decisions (for eDesign); conversely if there is no correlation between these rankings across requirements revision levels then the effects of correction/refinement decisions are greater than the effects of scoping decisions.

The following sections present the experiment setup and data, followed by analysis. Detailed results are provided in APPENDIX D.

#### *5.8.3.2 Experiment Setup*

For each DRA version under evaluation, a set of eleven dynamic properties was evaluated. The DRA versions were then ranked using property exception metrics as described in Section 3.2.13, Section 3.2.14, and Section 3.2.15. The properties and associated exception metrics that were used are listed in Table 21.

| PROPERTY CLASS | PROPERTY | METRICS |
|---|---|---|
| Correctness | Safety | $EXCP(C_S)$ |
| | Liveness | $EXCP(C_L)$ |
| | | |
| Performance | Usage Profile Latency | $EXCP(L_{UP})$ |
| | Usage Profile Throughput | $EXCP(TP_{UP})$ |
| | Component (DRAC) Utilization | $EXCP(U_C)$ |
| | Component (DRAC) Throughput | $EXCP(TP_C)$ |
| | Service Latency | $EXCP(L_{SVC})$ |
| | Service Utilization | $EXCP(U_{SVC})$ |
| | Service Throughput | $EXCP(TP_{SVC})$ |
| | | |
| Reliability | Component (DRAC) Reliability | $EXCP(R_C)$ |
| | Service Reliability | $EXCP(R_{SVC})$ |

Table 21 - Properties and Exceptions for Experiment 3

The evaluation methods and associated parameters used for evaluating this set of properties are listed in Table 22. Correctness evaluation parameters are described in Section 3.2.3, performance evaluation parameters are described in Section 3.2.7, and reliability evaluation parameters in Section 3.2.11.

| PROPERTY | METHOD | PARAMETERS |
|---|---|---|
| Correctness | Model Checking | MAX_ERRORS = 10 |
| | | |
| Performance | System Loading | IA_TIME = { 10, 20, 30, 40, 50, 60 , 70, 80 , 90, 100 } |
| | | TRIALS = 10 |
| | | SIM_TIME = 10000 |
| | | |
| Reliability | Sensitivity Eval. | $R_{ELEMp}$ = { 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 } |

Table 22 - Evaluation Methods and Parameters for Experiment 3

The threshold values for each of the property exceptions are listed in Table 23 (for definitions see Section 3.2.13, Section 3.2.14, and Section 3.2.15).

Property exceptions were evaluated for each DRA version over repeated trials. Results from all trials were collected and analyzed for statistically significant correlations.

| PROPERTY CLASS | PROPERTY | THRESHOLD |
|---|---|---|
| Correctness | Safety | $T(C_S) = 0$ |
| | Liveness | $T(C_L) = 0$ |
| | | |
| Performance | Usage Profile Latency | $T(L_{UP}) = 10.0$ |
| | Usage Profile Throughput | $T(TP_{UP}) = 0.33$ |
| | Component (DRAC) Utilization | $T(U_C) = 0.33$ |
| | Component (DRAC) Throughput | $T(T_C) = 0.33$ |
| | Service Latency | $T(L_{SVC}) = 10.0$ |
| | Service Utilization | $T(U_{SVC}) = 0.33$ |
| | Service Throughput | $T(TP_{SVC}) = 0.33$ |
| | | |
| Reliability | Component (DRAC) Reliability | $T(R_C) = 0.25$ |
| | Service Reliability | $T(R_{SVC}) = 0.25$ |

Table 23 - Threshold for Property Exceptions for Experiment 3

A statistical analysis of correlation was performed for the ranking results to test the following hypotheses:

$H_0$ - There is no correlation between relative DRA version rankings across requirements revision levels.

$H_1$ - There is a correlation between relative DRA version rankings across requirements revision levels.

The Kendall correlation coefficient was selected for this analysis [78, 97]. The reasons for choosing the Kendall correlation were (1) the experimental results represent categorical data with a non-normal distribution (thus eliminating the Pearson technique), and (2) the Spearman correlation coefficient is not well-suited for sample sizes of less than 30 (there were seven eDesign requirements revision levels). The Kendall correlation coefficient is computed as follows:

$$\tau = \frac{S}{\frac{1}{2}n(n-1)}$$

where:

`S` is a sum of scores

`N` is the number of elements

The degrees of freedom were computed as `v=5` (results were compared across seven eDesign requirements revision levels and the general formula for degrees of freedom for the Kendall coefficient is `v = N-2`). The confidence intervals associated with critical values of $\tau$ are summarized in Table 24 [97].

| CORRELATION ? | POSSIBLE | PROBABLE | VERY PROBABLE | ALMOST CERTAIN |
|---|---|---|---|---|
| Confidence Interval | 0.1000 | 0.0500 | 0.0200 | 0.0100 |
| Value of $\tau$ ($\pm$) | 0.6190 | 0.7143 | 0.8095 | 0.9048 |

Table 24 - Critical Values of $\tau$

Once a possible correlation was established to a confidence of 95% or higher, a two-tailed *p-value* was used to judge the significance of the Kendall coefficient. When the *p-value* was below a predetermined cut-off point, known as the significance level (formally expressed as the alpha, or $\alpha$-level), it was considered significant. The $\alpha$–level for this test was set to 0.05, therefore a p-value $\leq 0.05$ was considered to be significant (indicating a 95% confidence in the significance).

For each significant Kendall coefficient, a linear regression was performed to further determine the significance of the correlation. Correlations that exhibited a high degree of confidence and significance, but which did not yield good linear regression results were rejected. For this experiment, the $R^2$ value calculated by the linear regression was used to determine the goodness-of-fit. The $R^2$ statistic indicates how much of the variation within the sample is accounted for by the fitted regression line. Values closer to 1.0 indicate much variation has been accounted for by the regression. Table 25 summarizes the $R^2$ threshold used to judge goodness-of-fit for this experiment.

| GOODNESS-OF-FIT | WEAK | MODERATE | STRONG |
|---|---|---|---|
| $R^2$ Value | $0.5 < R^2 < 0.7$ | $0.7 < R^2 < 0.9$ | $0.9 < R^2$ |

Table 25 - Goodness-of-Fit Values for Experiment 3

#### *5.8.3.3 Experiment Data*

The experimental data set was formed by incrementally merging requirements to form DRA versions derived from various combinations of the six eDesign domain usage profiles for each of the seven eDesign requirements revision levels. For each requirements revision level, the six DRA versions were scoped according to the entries in Table 26.

| FUNCTIONAL SCOPES | USAGE PROFILE(S) INCLUDED |
|---|---|
| Scope 1 | Access Product Information |
| Scope 2 | Access Product Information<br>Publish New Product |
| Scope 3 | Access Product Information<br>Publish New Product<br>Publish New Collateral |
| Scope 4 | Access Product Information<br>Publish New Product<br>Publish New Collateral<br>Publish New Technical Document |
| Scope 5 | Access Product Information<br>Publish New Product<br>Publish New Collateral<br>Publish New Technical Document<br>Update Collateral |
| Scope 6 | Access Product Information<br>Publish New Product<br>Publish New Collateral<br>Publish New Technical Document<br>Update Collateral<br>Update Product |

Table 26- Stakeholder Priorities for Scoping Decisions

Table 27 summarizes the data used for this experiment. A total of 42 DRA versions were created. These DRA versions are grouped by their associated requirements revision levels.

| REQUIREMENTS REVISION LEVEL | DRA VERSION | FUNCTIONAL SCOPE (FROM TABLE 26) |
|---|---|---|
| REV0 | DRA1 REV0 | Scope 1 |
| | DRA2 REV0 | Scope 2 |
| | DRA3 REV0 | Scope 3 |
| | DRA4 REV0 | Scope 4 |
| | DRA5 REV0 | Scope 5 |
| | DRA6 REV0 | Scope 6 |
| REV1 | DRA1 REV1 | Scope 1 |
| | DRA2 REV1 | Scope 2 |
| | DRA3 REV1 | Scope 3 |
| | DRA4 REV1 | Scope 4 |
| | DRA5 REV1 | Scope 5 |
| | DRA6 REV1 | Scope 6 |
| REV2 | DRA1 REV2 | Scope 1 |
| | DRA2 REV2 | Scope 2 |
| | DRA3 REV2 | Scope 3 |
| | DRA4 REV2 | Scope 4 |
| | DRA5 REV2 | Scope 5 |
| | DRA6 REV2 | Scope 6 |
| REV3 | DRA1 REV3 | Scope 1 |
| | DRA2 REV3 | Scope 2 |
| | DRA3 REV3 | Scope 3 |
| | DRA4 REV3 | Scope 4 |
| | DRA5 REV3 | Scope 5 |
| | DRA6 REV3 | Scope 6 |
| REV4 | DRA1 REV4 | Scope 1 |
| | DRA2 REV4 | Scope 2 |
| | DRA3 REV4 | Scope 3 |
| | DRA4 REV4 | Scope 4 |
| | DRA5 REV4 | Scope 5 |
| | DRA6 REV4 | Scope 6 |
| REV5 | DRA1 REV5 | Scope 1 |
| | DRA2 REV5 | Scope 2 |
| | DRA3 REV5 | Scope 3 |
| | DRA4 REV5 | Scope 4 |
| | DRA5 REV5 | Scope 5 |
| | DRA6 REV5 | Scope 6 |
| REV6 | DRA1 REV6 | Scope 1 |
| | DRA2 REV6 | Scope 2 |
| | DRA3 REV6 | Scope 3 |
| | DRA4 REV6 | Scope 4 |
| | DRA5 REV6 | Scope 5 |
| | DRA6 REV6 | Scope 6 |

Table 27 - DRA Versions for Experiment 3

#### *5.8.3.4 Analysis and Conclusions*

The correlation results in APPENDIX D show a tendency towards positive correlations in how DRA versions were ranked relative to each other across requirements revision levels. This correlation data can be examined closer by computing the probability that a set of DRA versions ranked at a given requirements revision level will be ranked the same way for subsequent requirements revision levels. Based on the significant Kendall correlations, these probabilities are shown in Table 28 (note there are only six entries because requirements revision level REV6 has no successive revisions).

| REQUIREMENTS REVISION LEVEL | SIGNIFICANT CORRELATIONS TO SUCCESSIVE REVISION LEVELS | PROBABILITY OF CORRELATION |
|---|---|---|
| REV0 | 3 of 6 | 50% |
| REV1 | 1 of 5 | 20% |
| REV2 | 1 of 4 | 25% |
| REV3 | 0 of 3 | 0% |
| REV4 | 1 of 2 | 50% |
| REV5 | 1 of 1 | 100% |
| total | 7 of 21 | 33.33% |

Table 28 - Probability of Subsequent Correlations (95% Conf.)

Table 28 illustrates that in most cases there is a significant probability that a set of DRA versions ranked at a given requirements revision level will be ranked similarly for each subsequent requirements revision levels. This means that in many situations, the eDesign scoping decisions (e.g., selection of

functionality to include in a DRA version) had a greater impact on dynamic properties than the correction/refinement decisions (e.g., the requirements revision level had less impact on rankings than the scope of a DRA version). It is interesting to note that the degree of correlation seems to be affected by the number of correctness property errors present in the requirements revision level (this can be noted by the fact that the earliest revision levels had the greatest number of correctness errors, especially REV3 - see Section 4.3.1).

Table 29 shows a similar calculation for the probability that rankings will correlate between *consecutive* requirements revision levels. Here the effects of correctness property errors can be seen quite strongly as there was no probability of correlation across successive revision levels for early requirements revision levels (when the greatest number of correctness errors were being resolved).

| REQUIREMENTS REVISION LEVEL | PROBABILITY OF CORRELATION TO NEXT REVISION LEVEL |
|---|---|
| REV0 | 0% |
| REV1 | 0% |
| REV2 | 0% |
| REV3 | 0% |
| REV4 | 100% |
| REV5 | 100% |
| total | 33% |

Table 29 - Probability of Successive Correlations (95% Conf.)

The findings for the within-architecture decisions experiment are twofold. With respect to scoping decisions, the eDesign DRA versions showed a good probability of correlating DRA version rankings at a specific requirements revision level with rankings from subsequent requirements revision levels. This

means that, in general, there is a reasonable level of predictive power in the earliest evaluations, even in the presence of correctness errors. However, it is clear that correction/revision decisions can in many circumstances have a greater impact than the scoping decisions (as evidenced by the poor correlation for successive requirements revision levels at the earliest stages of requirements specification and revision). Therefore, it is likely that as requirements become more stable, scoping decisions have more impact than correction/refinement decisions. However, because the probability of correlation to future requirements revision levels is reasonably high there is still value in early evaluations, but re-evaluation must occur when requirements evolve as a result of the kinds of decisions examined in this experiment. These findings support the hypothesis for this experiment.

### 5.8.4 Experiment 4 - Across-Architecture Evolution

The hypothesis of this experiment is:

*Dynamic properties of early software architectures qualitatively change as a function of evolution across related architecture types.*

Across-architecture evolution in the SEPA 3D Architecture results in related families of architectures. Such families are created as the SEPA process is followed in defining DRA versions, AA versions, and IA versions, as described in

Section 2.2. The associations of these architectures are maintained by the mapping relations defined in Section 2.2.2 and Section 2.2.3.

The process begins when a DRA version is created to satisfy stakeholders' functional requirements. This DRA version becomes a starting point for a family of related architectures. Technology Solutions (TSs) are then selected to form an AA version in support of the DRA version. The TSs that form the AA version meet stakeholders' application requirements and provide the domain functionality specified in the associated DRA version. Additional TSs are then selected to form an IA version in support of (1) stakeholders' installation requirements, and (2) infrastructure demands of TSs selected for the associated AA version. Alternative TS selections for the AA or IA result in different AA or IA versions. A given DRA version can be associated with many related AA versions, each AA version specifying an alternative set of TSs to satisfy stakeholders' application requirements as well as the domain requirements in the associated DRA version. In a similar fashion, a given AA version may be associated with many IA versions.

Given this process, the intent of this experiment is to examine how these types of across-architecture decisions affect resulting dynamic properties. Therefore, the selected approach is to define a family of eDesign DRA, AA, and IA versions resulting from the various types of decisions described above, and to perform architecture ranking to illustrate the effects of the decisions (Section 3.2.12). The overall approach for this experiment is illustrated in Figure 60.

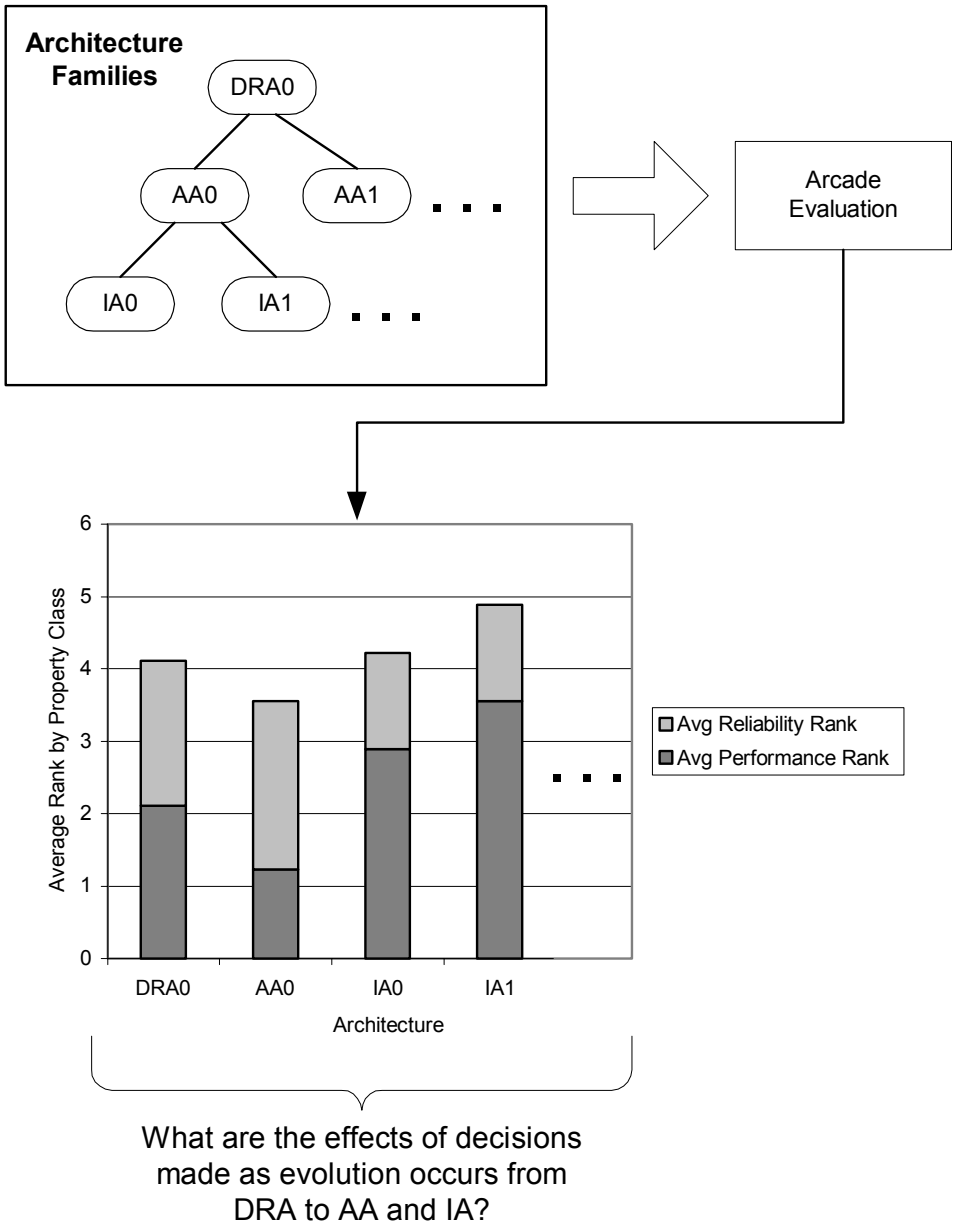


Figure 60 - Experimental Approach for Experiment 4

The following sections present the experiment setup and data, followed by analysis of the results. Details of results are provided in APPENDIX D.

#### *5.8.4.2 Experiment Setup*

Nine dynamic properties from the performance and reliability classes were evaluated using Arcade for each architecture version. Correctness was not evaluated because the architecture family was based upon $DRA_{REV6}$ which contained no correctness errors, and because this set of properties was sufficient to investigate how across-architecture decisions affect multiple properties from multiple property classes. The architecture versions were then ranked using property exception metrics as described in Section 3.2.12. The properties that were evaluated and the associated exception metrics are listed in Table 30.

| PROPERTY CLASS | PROPERTY | METRICS |
|---|---|---|
| Performance | Usage Profile Latency | $EXCP(L_{UP})$ |
| | Usage Profile Throughput | $EXCP(TP_{UP})$ |
| | Component (DRAC/TS) Utilization | $EXCP(U_C)$ |
| | Component (DRAC/TS) Throughput | $EXCP(TP_C)$ |
| | Service Latency | $EXCP(L_{SVC})$ |
| | Service Utilization | $EXCP(U_{SVC})$ |
| | Service Throughput | $EXCP(TP_{SVC})$ |
| | | |
| Reliability | Component (DRAC/TS) Reliability | $EXCP(R_C)$ |
| | Service Reliability | $EXCP(R_{SVC})$ |

Table 30 - Properties and Exceptions for Experiment 4

The evaluation methods and associated parameters used for evaluating this set of properties are listed in Table 31. Performance evaluation parameters are described in Section 3.2.7, and reliability evaluation parameters are described in Section 3.2.11.

| PROPERTY | METHOD | PARAMETERS |
|---|---|---|
| Performance | System Loading | IA_TIME = { 10, 20, 30, 40, 50, 60 , 70, 80 , 90, 100 } |
| | | TRIALS = 10 |
| | | SIM_TIME = 10000 |
| | | |
| Reliability | Sensitivity Eval. | $R_{ELEMp}$ = { 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 } |

Table 31 - Evaluation Methods and Parameters for Experiment 4

The threshold values for each of the property exceptions are listed in Table 32 (for definitions see Section 3.2.14, and Section 3.2.15). Property exceptions were evaluated for each architecture version over repeated trials.

| PROPERTY CLASS | PROPERTY | THRESHOLD |
|---|---|---|
| Performance | Usage Profile Latency | $T(L_{UP})$ = 10.0 |
| | Usage Profile Throughput | $T(TP_{UP})$ = 0.33 |
| | Component (DRAC) Utilization | $T(U_C)$ = 0.33 |
| | Component (DRAC) Throughput | $T(T_C)$ = 0.33 |
| | Service Latency | $T(L_{SVC})$ = 10.0 |
| | Service Utilization | $T(U_{SVC})$ = 0.33 |
| | Service Throughput | $T(TP_{SVC})$ = 0.33 |
| | | |
| Reliability | Component (DRAC) Reliability | $T(R_C)$ = 0.25 |
| | Service Reliability | $T(R_{SVC})$ = 0.25 |

Table 32 - Thresholds for Property Exceptions for Experiment 4

#### *5.8.4.3 Experiment Data*

The data for this experiment consists of a family of architectures associated with eDesign $DRA_{REV6}$. This family contains the original eDesign baseline DRA version, the original eDesign AA, and the original eDesign IA.

Alternative AA and IA versions were created to form the rest of the architecture family. . The resulting tree of thirteen architecture families is depicted in Figure 61. The experimental data set, including the types of across-architecture decisions that were associated with each architecture version, is summarized in Table 33.

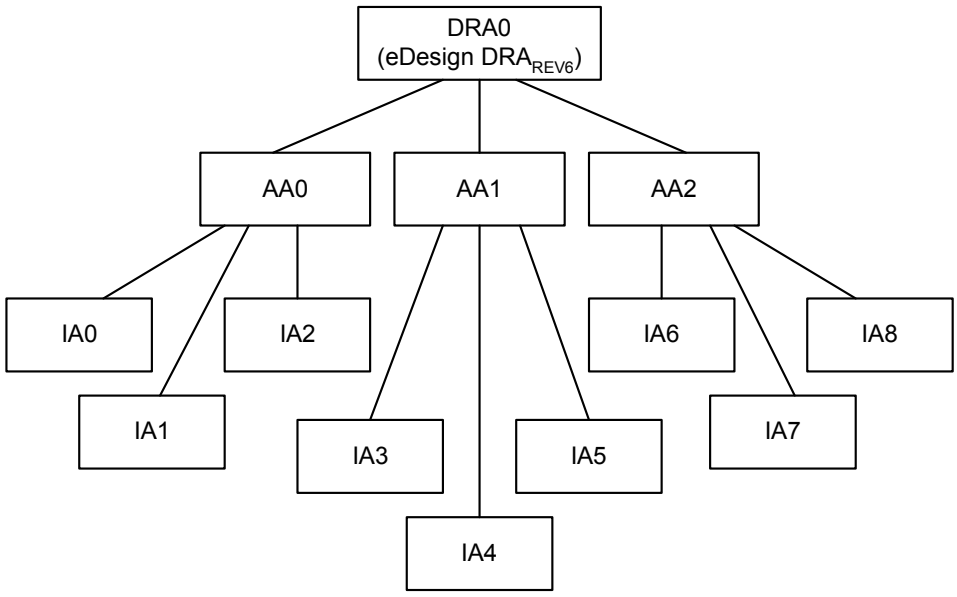


Figure 61- Architecture Family Tree for Experiment 4

| FAMILY | ARCHITECTURE | FORM OF ACROSS-ARCHITECTURE EVOLUTION |
|---|---|---|
| 1 | DRA0 | Baseline eDesign $DRA_{REV6}$ |
| | AA0 | TSs selected to support DRA0 and application requirements |
| | IA0 | TSs selected to support AA0 and installation requirements |
| | | |
| 2 | DRA0 | (described in Family 1) |
| | AA0 | (described in Family 1) |
| | IA1 | IA0 with modified compute environment execution speeds |
| | | |
| 3 | DRA0 | (described in Family 1) |
| | AA0 | (described in Family 1) |
| | IA2 | IA1 with modified communication channel latencies |
| | | |
| 4 | DRA0 | (described in Family 1) |
| | AA1 | TSs selected to support DRA0 and application requirements; AA1 ≠ AA0 |
| | IA3 | TSs selected to support AA1 and installation requirements |
| | | |
| 5 | DRA0 | (described in Family 1) |
| | AA1 | (described in family 4) |
| | IA4 | TSs selected to support AA1 and installation requirements; IA4 ≠ IA3 |
| | | |
| 6 | DRA0 | (described in Family 1) |
| | AA1 | (described in family 4) |
| | IA5 | TSs selected to support AA1 and installation requirements; IA5 ≠ IA4 ≠ IA3 |
| | | |
| 7 | DRA0 | (described in Family 1) |
| | AA2 | TSs selected to support DRA0 and application requirements; AA2 ≠ AA1 ≠ AA0 |
| | IA6 | TSs selected to support AA2 and installation requirements |
| | | |
| 8 | DRA0 | (described in Family 1) |
| | AA2 | (described in Family 7) |
| | IA7 | IA6 with modified compute environment execution speeds and communication channel latencies |
| | | |
| 9 | DRA0 | (described in Family 1) |
| | AA2 | (described in Family 7) |
| | IA8 | IA7 with modified compute environment execution speeds and communication channel latencies |

Table 33 - Experimental Data for Experiment 4

#### *5.8.4.4 Analysis and Conclusions*

The architecture family shown in Figure 61 has many related architectures. Each branch in the tree represents alternative architectural decisions to meet a set of requirements. For example, AA0, AA1, and AA2 all satisfy (1) the stakeholder functional requirements specified by DRA0, and (2) stakeholder application requirements. Yet each AA version resulted from different across-architecture decisions. The hypothesis of this experiment states that these decisions will result in detectable differences in the dynamic properties of the architecture versions.

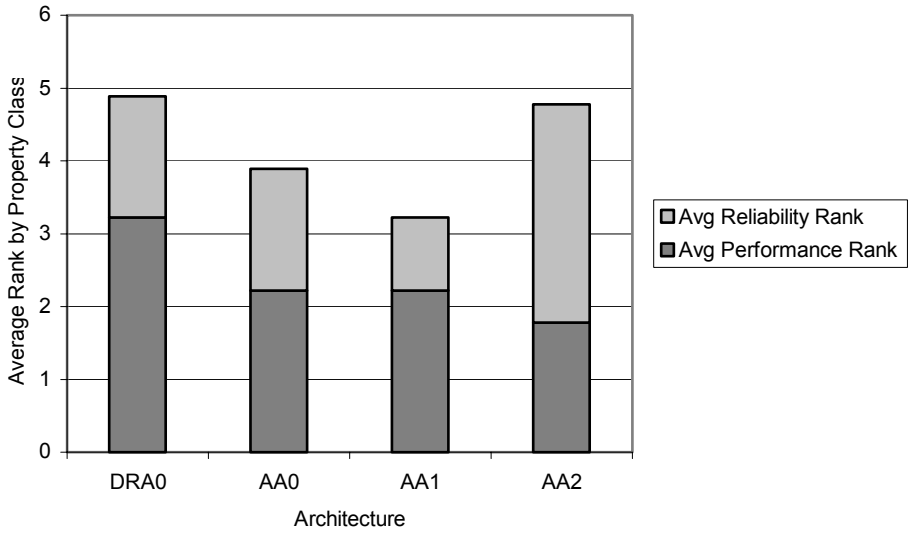


Figure 62 - Rankings for Alternative AA Versions

Figure 62 shows the rankings for the three AA versions associated with DRA0. Here it can be seen that AA1 has perhaps the best tradeoffs between

performance and reliability properties. The across-architecture evolution decisions regarding allocation of functionality to Technology Solutions (TSs) by all of the AA versions in Figure 62 achieved better performance than the allocation of functionality to DRACs suggested by DRA0, while AA2 showed significant reliability degradation.

Examining across-architecture evolution between AA versions and IA versions provides additional interesting observations. Graphs for each branch of the architecture tree with unique AA versions are shown in Figure 63 though Figure 65 (for example, "Branch 1" is that part of the tree in Figure 61 containing DRA0, AA0, IA0, IA1, and IA2). In Figure 65 it can be seen that in every case the across-architecture decisions made for IA versions had a negative impact on performance and a positive impact on reliability with respect to both DRA0 and AA0.

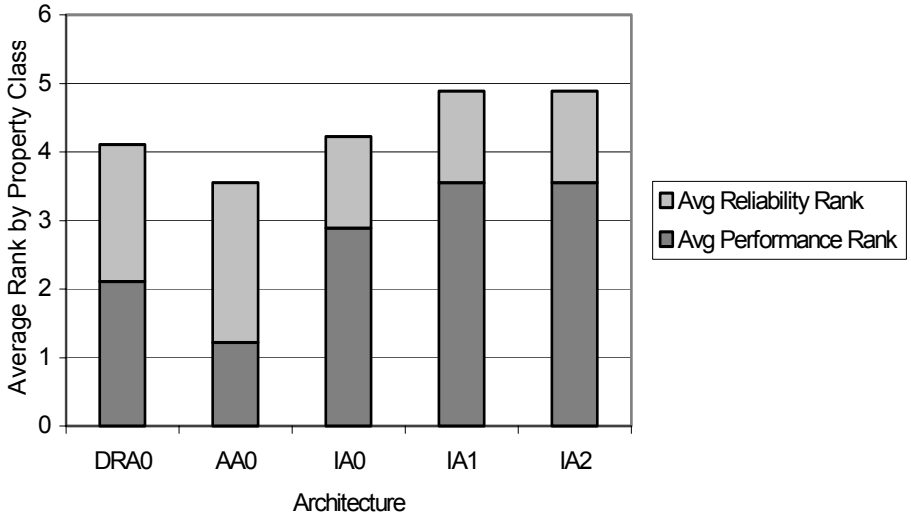


Figure 63 - Rankings for Implementation Architectures (Branch 1)

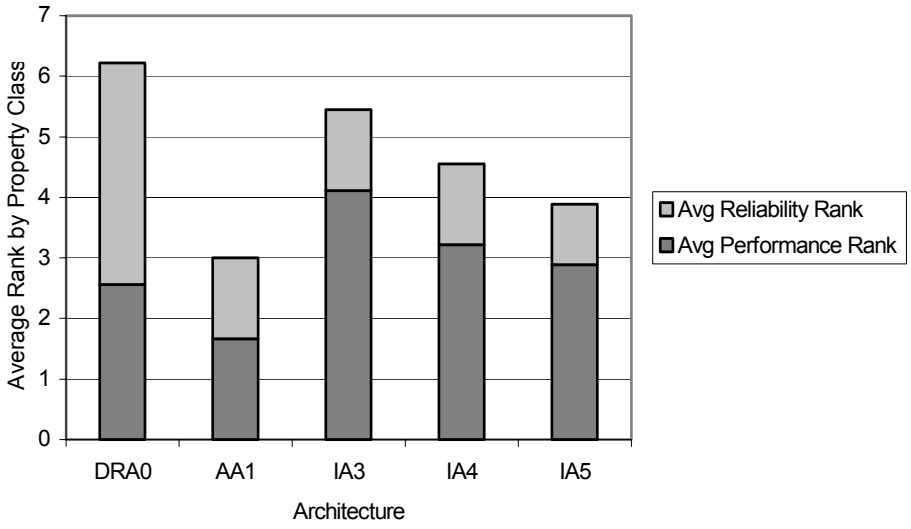


Figure 64 - Rankings for Implementation Architectures (Branch 2)

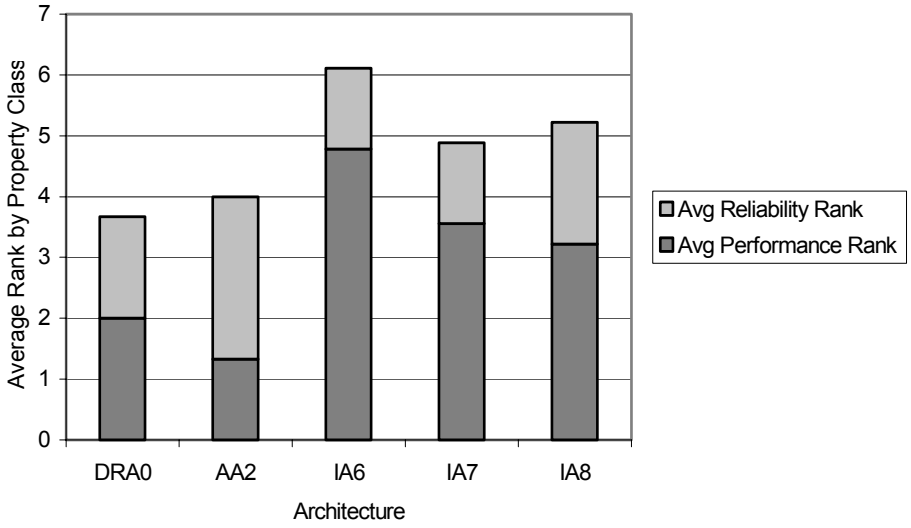


Figure 65 - Rankings for Implementation Architectures (Branch 3)

The negative changes in performance and the positive changes in reliability in Figure 63 through Figure 65 can be explained as follows. In every case of DRA and AA evaluation, performance is calculated as if the components and services were executing in an ideal environment (e.g., all execution speeds relative to each other are equivalent, and all communication channel latencies are infinitely small). Similarly, reliability is calculated as if each service or component were running in an isolated compute environment. These assumptions are necessary because the DRA and the AA contain no information regarding computing environments or communications channels. Decisions related to computing environments and communications channels are only made when an IA version is created. The net effect of the IA decisions made in this experiment was to introduce the AA and DRA elements into more constrained compute environments (e.g., execution speeds of compute environments and communication channel latencies became less than ideal with respect to the DRA and AA). The impact of these decisions on performance properties was generally negative. For reliability, the assumption that each service or component executed in its own environment was changed by decisions to group these elements into sets of common environments, resulting in less inter-environment complexity and a corresponding net improvement in reliability.

Figure 66 shows rankings of the entire architecture family tree. From this figure, it can be seen that the best overall branch of the tree was the branch that contained AA1 and that the best IA version from this branch was IA5.

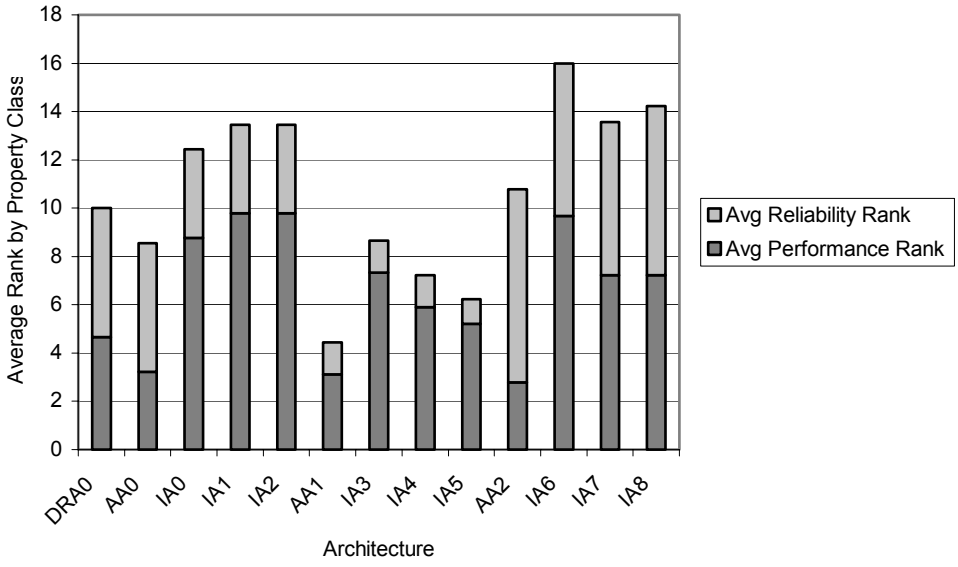


Figure 66 - Rankings for Entire Architecture Tree

The results of this experiment offer good evidence that the effects of across-architecture decisions for early architecture versions can be detected and explained with the Arcade property exception ranking techniques.

## 5.9 RESEARCH QUESTION #5

*Can the rapid feedback from software architecture execution evaluations serve to influence analysis and design decisions in an iterative software engineering process?*

A combination of implementation, experimentation, and the eDesign case study were performed in support of this question. The implementation task involved creating a feedback mechanism between Arcade and RARE. Experimentation was then performed to determine whether feedback from Arcade to RARE could influence analysis and design decisions. The eDesign case study is discussed in Chapter CHAPTER 4. RARE and the Arcade-to-RARE feedback mechanism are described in Section 3.1.5. The following sections present the experiment performed in support of this research question.

### 5.9.1 Experiment 5 - RARE Feedback

The hypothesis of this experiment is:

> *Rapid feedback of dynamic property evaluation results to the architecture derivation process can improve the degree to which the derivation process can satisfy stakeholder goals with respect to dynamic properties.*

Because RARE can only supply static property metrics to its heuristics, it was desirable to determine whether the availability of dynamic property metrics could enhance RARE's ability to perform derivations that meet stakeholders' goals for both static and dynamic properties. Therefore, a mechanism was created whereby Arcade evaluation results could be fed back to RARE during derivation [15]. Using this mechanism, experimentation was performed via the following steps: (1) use RARE to produce an eDesign DRA version containing all eDesign functionality, (2) use Arcade to evaluate performance properties of this DRA version, (3) feed results back to RARE for use in deriving a second DRA version, this time employing the performance feedback with additional performance heuristics, (4) repeat Arcade evaluation for the new DRA version, and (5) analyze the difference in performance metrics between the two DRA versions to understand whether RARE was able to make effective use of the performance metrics fed back from Arcade. The overall evaluation and feedback process used for this approach was described and illustrated in Section 3.1.5.

The following sections present the experiment setup and data, followed by analysis of the results.

#### *5.9.1.1 Experiment Setup*

A set of seven performance properties was evaluated for this experiment. This set of properties is listed in Table 34.

| PROPERTY CLASS | PROPERTY | METRIC |
|---|---|---|
| Performance | Usage Profile Latency | $\overline{M(L_{UP})}$ for repeated trials |
| | Usage Profile Throughput | $\overline{M(TP_{UP})}$ for repeated trials |
| | Component (DRAC) Utilization | $\overline{M(U_C)}$ for repeated trials |
| | Component (DRAC) Throughput | $\overline{M(TP_C)}$ for repeated trials |
| | Service Latency | $\overline{M(L_{SVC})}$ for repeated trials |
| | Service Utilization | $\overline{M(U_{SVC})}$ for repeated trials |
| | Service Throughput | $\overline{M(TP_{SVC})}$ for repeated trials |

Table 34- Properties Evaluated in Experiment 5

Specific dynamic property evaluation methods and associated Arcade evaluation parameter settings are listed in Table 35. Arcade's performance evaluation parameters are defined in Section 3.2.7.

| PROPERTY | METHOD | PARAMETERS |
|---|---|---|
| Performance | System Loading | `IA_TIME` = { 10, 20, 30, 40, 50, 60 , 70, 80 , 90, 100 } |
| | | `TRIALS` = 10 |
| | | `SIM_TIME` = 10000 |

Table 35 - Evaluation Methods and Parameters for Experiment 5

For the second DRA derivation iteration, RARE was configured to focus on improving the performance of the eDesign "Access Product Information" usage profile. This usage profile was identified as a performance-critical usage profile by stakeholders, and was also a focus of performance evaluation during the eDesign case study (Section 4.3.2).

#### *5.9.1.2 Experiment Data*

The input data for this experiment was eDesign $DRA_{REV6}$ (Section 5.1).

#### *5.9.1.3 Analysis and Conclusions*

The graph in Figure 67 shows $L_{UP}$ (usage profile latency) of the "Access Product Information" usage profile for both $DRA_{REV6}$ and the new DRA version ($DRA_{REV6+FDBK}$) derived using Arcade feedback.

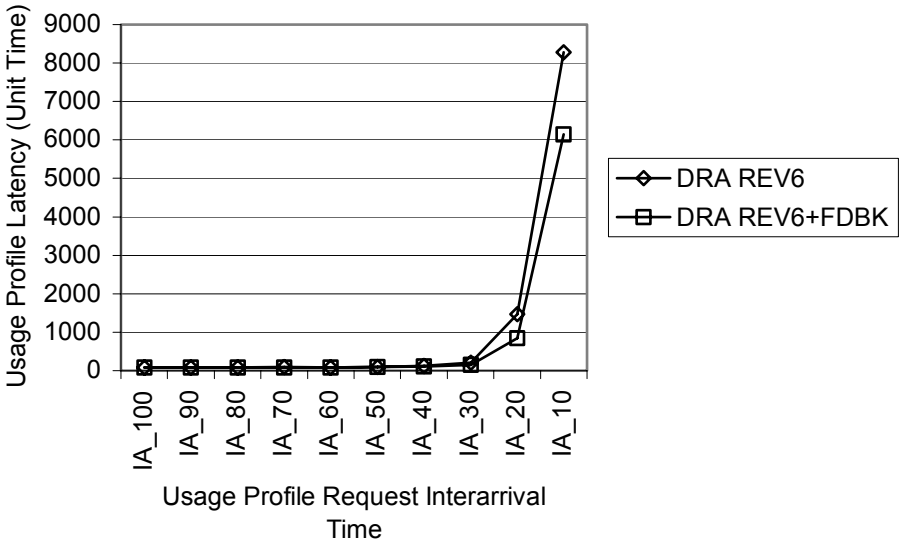


Figure 67 - Usage Profile Latency for Experiment 5

Table 36 summarizes the percent improvement in the "Access Product Info" usage profile across the DRA versions. The mean improvement was approximately 12% across all system loadings.

| | IA_100 | IA_90 | IA_80 | IA_70 | IA_60 | IA_50 | IA_40 | IA_30 | IA_20 | IA_10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $L_{UP}$ (Baseline DRA) | 83.67 | 85.12 | 87.09 | 90.31 | 88.41 | 100.7 | 128.01 | 202.49 | 1473.35 | 8278.65 |
| $L_{UP}$ (Modified DRA) | 79.94 | 81.26 | 85.09 | 87.48 | 87.63 | 103.48 | 113.09 | 154.41 | 844.72 | 6138.57 |
| %improvement | 4% | 5% | 2% | 3% | 1% | -3% | 12% | 24% | 43% | 26% |

Table 36 - Percent Improvement for Usage Profile Latency

Figure 68 illustrates the exchange between static and dynamic evaluation in RARE and Arcade in this experiment. RARE was configured to emphasize "performance," "reusability," and "maintainability" qualities, prioritized in that order. Given the selected qualities, RARE generated a DRA consisting of five DRACs. Based on the priority of the "performance" goal, heuristic "Reduce DRAC coupling to decrease inter-DRAC communication" determined the primary structure of the DRA in "pass n," and success of this heuristic is determined in part by the static structural metric, "Coupling Factor," which measures the degree to which DRACs are coupled through data dependencies, implying the need for communication channels. Through performance evaluation, Arcade generated feedback usable by RARE for subsequent refinement: "DRAC Utilization at IA 10" and "Service Utilization at IA 10." In the following pass, RARE determined the value of "DRAC Utilization at IA 10" for DRAC "Customer" to be unacceptable, triggering performance heuristic "Redistribute services with high utilization to reduce overall DRAC utilization." Examining values for "Service Utilization at IA 10," a strategy under this heuristic created DRAC "Customer -2" and identified service "Download Product Collateral" as a candidate to move

from DRAC "Customer." Arcade evaluation then proceeded with the new six-DRAC architecture.

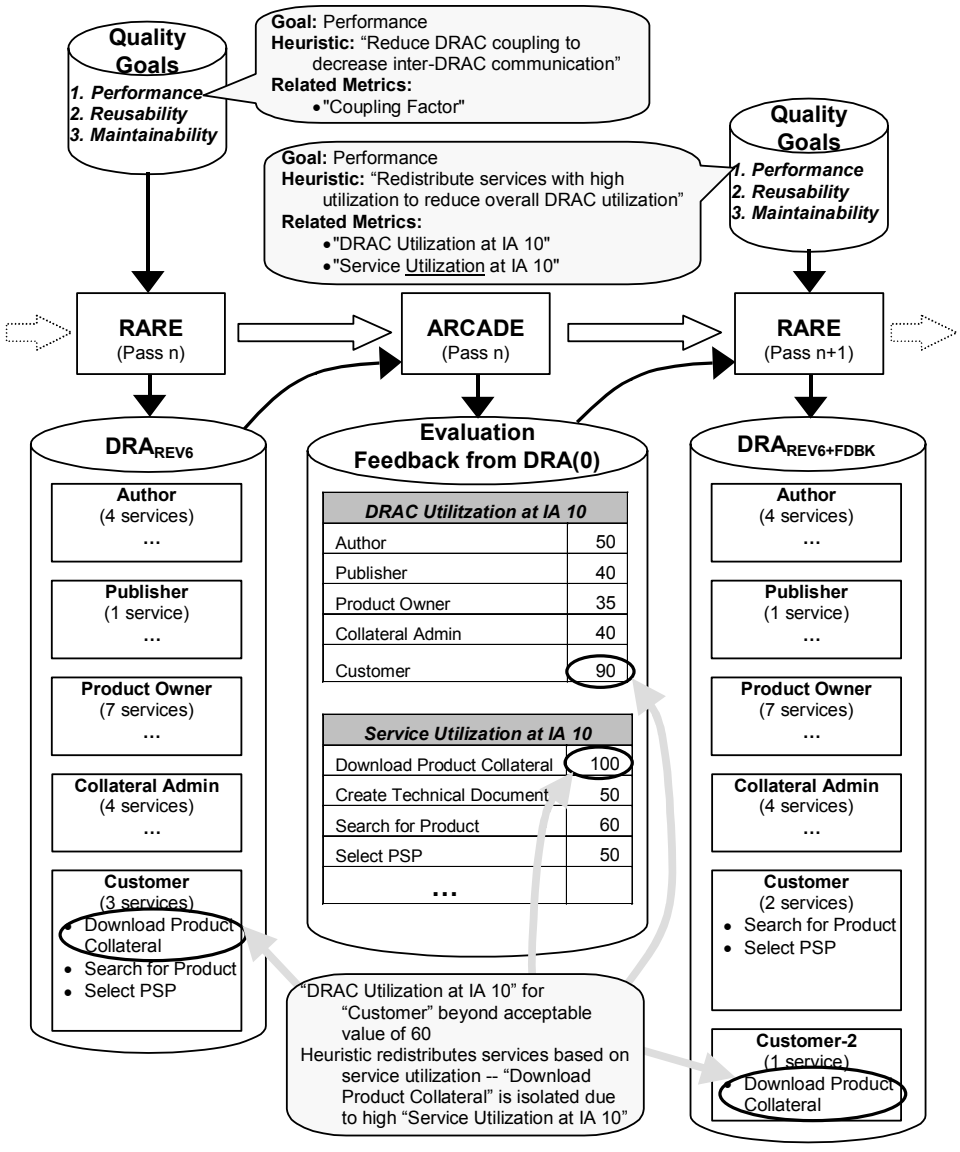


Figure 68 - The Feedback Process in Experiment 5

The information that Arcade supplied to RARE was created quickly and automatically. RARE readily applied the additional information to make structural allocation decisions that resulted in a mean improvement of approximately 12% for the "Access Product Information" usage profile. For heavy system loadings (`IA_TIME` = {40, 30, 20, 10}) the mean improvement was approximately 26%.

These experimental results provide evidence that the rapid feedback of the Arcade dynamic property evaluation results had a positive effect in allowing RARE to make adjustments to an architecture. The feedback process ultimately improved the latency metrics of a critical usage profile by altering RARE's suggested allocation of domain functionality amongst DRACs. This evidence supports the hypothesis for experiment 5.

# CHAPTER 6 CONCLUSIONS

This dissertation presented a program of research and experimentation that examined processes and techniques for early and iterative evaluation of dynamic properties of requirements. Specifically, the focus of this research was using software architectures as requirements models in conjunction with software architecture execution techniques for dynamic property evaluations.

The following sections discuss the research contributions and future work.

## 6.1 CONTRIBUTIONS

The main contributions resulting from the research and experimentation associated with this dissertation are:

- **A systematic process and supporting tool for early and iterative evaluation of dynamic properties;**
- **A set of experimental results in support of the research hypothesis and associated research questions; and**
- **A case study involving application of the process and tool in an industrial software development project.**

Each of these contributions is summarized in the following sections.

### 6.1.1 Systematic Process and Supporting Tool

The dynamic property evaluation process demonstrated in this dissertation can be systematically employed to: (1) evaluate dynamic properties early and iteratively, and (2) allow stakeholders who are not experts in software architecture execution to perform evaluations. The Arcade tool was developed to support this dynamic property evaluation process.

Arcade leverages the SEPA 3D Architecture in conjunction with a number of dynamic property evaluation techniques, including model checking, discrete event simulation, and probabilistic graph model algorithms. In doing so, Arcade addresses several practicality issues associated with the selected evaluation techniques. These practicality issues can be divided into expertise-related issues and capacity-related issues.

Expertise-related issues addressed by Arcade are: (1) automated translation of requirements specified in a software architecture model into a format suitable for dynamic property evaluation techniques, and (2) automated collection and presentation of evaluation results to stakeholders in intuitive formats. Thus, Arcade enables stakeholders to perform dynamic property evaluations, to understand the evaluation results, and to make decisions based upon those results.

Arcade addresses capacity-related issues by leveraging partitioned software architecture models (e.g., the SEPA 3D Architecture) to reduce the complexity of evaluation. This is important because complexity affects: (1) the human capacity to produce and work with large models and associated evaluation

results, and (2) computer capacity issues such as CPU and memory requirements. Arcade's support for partitioned software architecture models mitigates complexity by allowing dynamic property evaluation to be performed early (e.g., using partial requirements specifications), and iteratively (e.g., using an incremental approach to requirements modeling and evaluation).

The ability to partition requirements types amongst the SEPA DRA, AA, and IA is an important enabler for early, iterative dynamic property evaluation. For example, because the DRA is a highly abstract representation of functionality that is independent of any particular implementation, DRA evaluation can uncover errors associated with domain requirements early in the requirements modeling and analysis process. Subsequently, the AA and IA can be used to evaluate design tradeoffs and site configuration implications once customer site requirements are known.

Arcade - in conjunction with the SEPA 3D Architecture - supports requirements partitioning in two dimensions. The first requirements partitioning dimension is defined by the SEPA 3D Architecture: partitioning across requirements types (e.g., the DRA, AA, and IA models). This partitioning dimension enables both early evaluation and subsequent reuse of evaluation results. For example, evaluating only the requirements specified in a DRA reduces the complexity of the behavioral specification by expressing system behavior at an abstract domain level (e.g., business process functionality, data and their relationships, timing between functions), while deferring evaluation of requirements specified in the AA and IA. The second requirements partitioning

dimension is defined by partitioning within a single requirements type to create a partial model. This partitioning dimension is a key enabler for allowing evaluations to proceed concurrently with requirements acquisition and evolution. For example, a partial model of a DRA might represent only the domain requirements necessary to accomplish a single domain process, thereby focusing evaluation on specific functionality of interest early during the requirements acquisition and modeling process, before requirements are completely modeled. In addition to mitigating complexity, the incremental approach supported by these partitioning dimensions can enable the ability to evaluate the impact of requirements evolution.

### 6.1.2 Experimental Results

The experimentation associated with this dissertation was performed in support of the following hypothesis:

> **Software Architecture Execution will allow for a systematic, automated evaluation of non-static (dynamic) architectural properties to assess dynamic properties in isolation, dependencies between dynamic properties, impact of early analysis and design decisions including the architectural derivation process and architecture specification content.**

Five experiments were conducted to investigate various aspects of this hypothesis. The contributions of these experiments are summarized in the following sections.

#### *6.1.2.1 Experiment 1 - Dynamic Property Dependencies*

The goal of this experiment was to investigate whether dynamic property dependencies could be detected using early software architecture artifacts and software architecture execution techniques. Therefore, the following hypothesis was defined for Experiment 1:

> *Early software architecture artifacts can support evaluation of dynamic property dependencies using software architecture execution.*

A total of 35 architectures were evaluated for a set of 11 dynamic properties (Section 5.6.1). Statistical analysis was performed on the evaluation results to determine whether significant correlations between dynamic properties could be detected using Arcade. It was determined that about 8 of the 55 pairs of dynamic properties showed a statistically significant correlation. These experimental results support the hypothesis that early software architecture artifacts contain sufficient information to detect dynamic property dependencies.

#### *6.1.2.2 Experiment 2 - Effects of Structural Decisions*

The goal of this experiment was to investigate whether dynamic properties of early software architecture artifacts are affected by structural decisions (e.g., the allocation of functionality to structure). Therefore, the following hypothesis was defined for Experiment 2:

> *Early software architecture artifacts can support evaluation of the effects of structural decisions on dynamic properties.*

Evaluations of nine dynamic properties were performed for eleven DRA versions (Section 5.8.1). This set of architectures consisted of structural allocations for all combinations of functionality associated with six eDesign usage profiles to one, two, and three DRACs. The DRAs were then ranked using the comparison techniques developed in this dissertation (Section 3.2.12). The experimental results show the structural decisions regarding allocation of services to DRACs produced a consistent and explainable pattern of rankings based on individual dynamic properties across the various architectures. The variance of the property exception metrics was moderate, establishing confidence that computing the number of property exceptions is a reasonable means of ranking architectures. Furthermore, the cumulative ranking techniques (Section 3.2.16) were also useful in understanding the effects of different structural allocations on resulting dynamic properties of the architectures. These results provide good

support for the experimental hypothesis, showing that the effects of structural decisions can be evaluated for early software architecture artifacts.

#### *6.1.2.3 Experiment 3 - Within-Architecture Evolution Decisions*

Within-architecture evolution decisions affect a single architecture type (e.g., the SEPA DRA, AA, or IA). This experiment examines the effects of two types of within-architecture decisions on dynamic properties: *scoping decisions* whereby the scope of functionality included in an architecture version is determined, and *correction/refinement decisions* whereby the details of functional and non-functional requirements in an architecture are adjusted (Section 5.8.3). The following hypothesis was defined for Experiment 3:

> *Dynamic properties of early software architectures qualitatively change as a function of evolution within the architecture.*

A total of 42 DRA versions were evaluated in this experiment. The experimental data were grouped so that the effects of both scoping decisions and correction/refinement decisions could be evaluated. The findings of this experiment were that in the early stages of architecture evolution the correction/refinement decisions were likely to have a greater impact on dynamic properties. Subsequently, scooping issues have a greater impact on dynamic properties. These findings support the hypothesis, illustrating that the effects of

within-architecture evolution decisions on dynamic properties can be evaluated for early software architecture artifacts.

#### *6.1.2.4 Experiment 4 - Across-Architecture Evolution Decisions*

Across-architecture decisions involve mapping and associating elements of one architecture type to elements of another architecture type. These decisions can affect the allocation of functionality to structure, the allocation of software to hardware, as well as non-functional requirements such as speed of compute environments, or latency of communications channels. Therefore, the following hypothesis was defined for Experiment 4:

> *Dynamic properties of early software architectures qualitatively change as a function of evolution across related architecture types.*

This experiment evaluated a set of thirteen architectures that comprised nine distinct architecture families (Section 5.8.4). Each family was composed of a DRA version, an AA version, and an IA version. The comparison techniques developed in this dissertation (Section 3.2.12) were applied to understand the effects of decisions that resulted in different AA and IA versions associated with a single DRA version. The evaluation produced distinct patterns of architecture rankings based on the decisions made (1) as across-architecture evolution occurred from the DRA to the AA, and (2) as across-architecture evolution

occurred from the AA to the IA. These patterns were explainable by examining the types of decisions that resulted in the associated architecture versions. Furthermore, tendencies in dynamic property metrics that were established by across-architecture decisions relating a DRA version to an AA version could be seen to be preserved following decisions that related that AA version to an IA version (e.g., improvements or degradations in dynamic properties of an AA version with respect to its associated DRA version were preserved in a relative fashion for IA versions that were associated with that AA version). This evidence supports the experimental hypothesis for Experiment 4.

#### *6.1.2.5 Experiment 5*

The goal of this experiment was to investigate using the results of Arcade evaluations to affect the decisions made during architecture specification. Therefore, the following hypothesis was defined for Experiment 5:

> *Rapid feedback of dynamic property evaluation results to the architecture derivation process can improve the degree to which the derivation process can satisfy stakeholder goals with respect to dynamic properties.*

The approach for this experiment was to employ the results of Arcade performance evaluations in the RARE derivation process (Section 5.9.1). RARE is a research tool that derives a Domain Reference Architecture (DRA, Section

2.2.1) using heuristics based on static attributes of an architecture. By supplementing static information with information gained from dynamic property evaluations, RARE is able to employ additional heuristics during the derivation process. The results of this experiment indicate that early Arcade performance evaluations can produce feedback that enables RARE to subsequently improve the performance of selected elements of the architecture. A DRA created using the feedback process represents an architectural blueprint that incorporates the benefits of both static and dynamic evaluations.

### 6.1.3 Case Study

The eDesign stakeholders employed Arcade's qualitative evaluation process for several valuable purposes, including (1) aiding in detection and correction of requirements errors, (2) understanding the effects of requirements evolution, and (3) providing guidance to system implementers.

Arcade's incremental approach was effective for mitigating complexity while discovering correctness errors in the eDesign DRA. Correctness evaluation provided the opportunity to resolve functional errors (e.g., correction of completeness, safety, and liveness errors) before significant architecture commitments had been made. Project participants acknowledged the cost savings in correcting errors in the DRA specification rather than correcting errors detected after system implementation. Subsequent evaluation of the eDesign AA and IA yielded additional information concerning the effects of non-functional requirements and installation requirements on performance and reliability

properties of the system. This information was useful to eDesign stakeholders in helping them to understand the effects of their implementation and deployment decisions.

The eDesign case study illustrates how Arcade effectively uses the SEPA 3D Architecture to help manage complexity, to reduce the level of expertise required to perform dynamic property evaluations, and to support an iterative approach allowing early, incremental evaluation using partial models. While a moderate initial investment in time and training was required to utilize the Arcade approach, the eDesign team was subsequently able to gain valuable insight from correctness, performance, and reliability evaluations, for verifying requirements specified in the architecture and providing rationale for subsequent design and implementation decisions that would have otherwise been supported solely by intuition.

## 6.2 FUTURE WORK

A fundamental aspect of the Arcade approach is reducing the need for tool expertise by providing a layer of automation between well-defined representations appropriate for early requirements specification and sophisticated evaluation techniques capable of offering valuable insight. However, as with any attempt to render an approach more easily adopted, reducing the need for expertise comes at the cost of limiting access to advanced features of underlying evaluation tools. For example, non-expert users who have no knowledge of model checkers and associated languages employed by Arcade (e.g., SPIN and Promela) can use

Arcade to verify automatically defined correctness properties, but will not have the knowledge required to manually define additional properties for verification. While it is possible for an expert user to employ Arcade artifacts (for example the Promela code generated by Arcade) as a starting point for advanced evaluations using underlying evaluation tools directly, more research is needed to determine the extent to which advanced features can be made directly available to developers without requiring an unacceptable level of training and expertise, thereby reducing the tool's practical appeal.

With the SBRA reliability evaluation technique, the probability of interaction between architecture elements (e.g., services, components, or compute environments) has a direct correlation to the influence of those element's reliability on the reliability of the entire architecture. However, while certain elements may interact with low frequency, they may still be critical to the operation of the architecture as a whole. For example, in the eDesign case study, no usage profiles could succeed if the usage profile "Publish New Technical Document" had not succeeded at least one time. These dependencies between the successful execution of usage profiles are not explicitly modeled in SBRA. Future research could investigate addressing this issue by incorporating the failure mode analysis and risk assessment techniques that have recently been layered on top of SBRA [118].

Another area of research that warrants additional investigation is that of early and iterative feedback between static and dynamic evaluation or architecture properties. The results of Experiment 5 (Section 5.9) offer promise that this effort

could yield valuable results. Automated support of such a process could enable stakeholders to rapidly explore a full set of properties for alternative architectures. Such rapid evaluation could help stakeholders make reasoned tradeoffs between many competing architectural priorities, and the resulting architectures would have a supporting rationale based upon predicted values of system properties.

# APPENDIX A - THE SEPA DOMAIN REFERENCE ARCHITECTURE

This appendix contains a formal definition of the SEPA Domain Reference Architecture (DRA). The DRA is an architecture-based requirements model for representing domain requirements. For all subsequent formal definitions, the following notational conventions will be used. Structural elements in the DRA metamodel will be defined in the form `x:`*`Type`* or `x:{`*`Type`*`}` which declares an element named `x` of type *`Type`* or a collection of elements of type *`Type`*, respectively. Types of elements include DRACs, services, and attributes. Elements declared of type *`Type`* may contain one or more named fields, where each field is declared as a particular element type or a collection of elements of that type. An individual field under a declared element can be referenced through the dot operator ("`.`"). For instance, field $x_1$ of declared element `x` can be referenced using the expression `x.`$x_1$. If an element belongs to a specific collection defined elsewhere in the representation, that is noted accordingly (e.g., $x \in X$).

The definitions to follow will also utilize the `refArch(x)` operator, which is often useful to reference the DRA to which an element belongs (i.e., `refArch(x)` identifies the DRA to which `x` belongs).

**DRA Representation**

The Domain Reference Architecture (DRA) can be described by the 5-tuple, $(n, r, \theta_r, D_{RA}, S_{RA})$, shown in Figure 69. A DRA is uniquely identified by a name and a release number, `ra.n` and `ra.r`, respectively.

| $ra : DRA \triangleq (n, r, \theta_r, D_{RA}, S_{RA})$ | |
|---|---|
| $n : String$ | name of the domain reference architecture. |
| $r : \mathbb{N}$ | architecture release number. |
| $\theta_{ra} : Trc$ | domain reference architecture traceability. |
| $D_{ra} : \{DRAC\}$ | set of all DRACs in Domain Reference Architecture RA. |
| $S_{RA} : \{Subsys\}$ | set of all subsystems in Domain Reference Architecture RA. Each subsystem is composed of selected DRACs. |
| $s : Subsys = (n, \theta, D_s) \in S_{RA}$ where | |
| $n : String$ | name of the subsystem. |
| $\theta : Trc$ | subsystem traceability. |
| $D_s : \{DRAC\} \subseteq D_{RA}$ | DRAC members of subsystem. |

Figure 69 - DRA Representation

Each DRA has associated collections of DRACs, $ra.D_{ra}$, and DRAC subsystems, $ra.S_{ra}$. The representation for each DRAC in $ra.D_{ra}$ will be defined in the following section. Subsystems referenced by $ra.S_{ra}$ are collections of DRACs in the DRA based on a variety of justifications, including, but not limited to (1) domain tasks typically co-located; (2) collections of domain tasks provided by legacy applications; and (3) collections of strongly-coupled domain tasks (i.e., share data and events). These justifications reflect the architect's intentions introduced during the architecture derivation process.

**DRAC Representation**

A DRAC is represented as the 5-tuple, $(n, \theta_d, A_d, S_d, B_d)$, shown in Figure 70. Each DRAC is identified by a unique name in the DRA ($d.n$), and collections of attributes ($d.A_d$) and services ($d.S_d$) comprise the Declarative Model (DRAC D-M). A DRAC's Behavioral Model (DRAC B-M) , $d.B_d$, is represented by a high-level hierarchical state chart (i.e., states, transitions between states, guards, and events). The Integration Model (DRAC I-M) is not represented explicitly. Rather, the dependency information provided by the DRAC I-M is derived from (1) the inputs and outputs described in DRAC D-M services where the inputs/outputs cross DRAC boundaries and (2) the subsystems defined in the DRA. The DRAC D-M, DRAC B-M, and DRAC I-M are described below.

| $d: DRAC \stackrel{def}{=} (n, \theta_d, A_d, S_d, B_d) \in ra.D_{ra}$ | |
|---|---|
| $n: String$ | name of the DRAC. The name must be unique among all DRACs in the respective DRA. |
| $\theta_d: Trc$ | DRAC traceability. |
| $A_d: \{Attr\}$ | set of attributes owned by DRAC $d$. This information belongs to the Declarative Model. |
| $S_d: \{Svc\}$ | set of services provided by DRAC $d$. This information belongs to the Declarative Model. |
| $b_d: StChart$ | high-level statechart representing the DRAC $d$ Behavioral Model. |

Figure 70 - DRAC Representation

**DRAC Declarative Model (DRAC D-M) Representation**

A DRAC's D-M is comprised of (1) attributes representing domain data and events owned by the DRAC (Figure 71) and (2) services representing domain tasks offered by the DRAC (Figure 72). Each attribute owned by a DRAC, $a \in d.A_d$, can be described by the 5-tuple, $(n, \theta_a, y, k, A_c)$, in Figure 71. Attributes are directly traceable to domain data concepts and domain events. As such, the attribute type, $a.y$, characterizes an attribute as either an event or a data concept; data types include one of the following: (1) a primitive data type (e.g., string, integer, or float), (2) "composite," or (3) "unspecified." A "composite" data attribute is "composed of" other data attributes. For example, a technical document is likely composed of multiple fields, such as a title, keywords, and

body. The constituent attributes for a "composite" attribute are referenced through the field $a.A_c$. Constituents must belong to the same DRAC as the composite, and attributes may not be recursively composed.

| $a:Attr \stackrel{def}{=} (n,\theta_a,y,k,A_c) \in d.A_d$ | |
|---|---|
| $n:String$ | name of the attribute. |
| $\theta_a:Trc$ | attribute traceability. Attributes can represent either domain data concepts or domain events. |
| $y$ | type of the attribute. If the attribute is composed of other attributes it carries a type of "composite."<br>$a.y \in \{Event,String,Integer,$<br>$Float,Composite,$<br>$Unspecified\}$ |
| $k$ | cardinality of the attribute as it typically exists in the domain or under its parent if it is a constituent of that parent. ("Event" attributes always have cardinality of "one.")<br>$a.k \in \{one,zero\text{-}to\text{-}one,$<br>$zero\text{-}to\text{-}many,$<br>$one\text{-}to\text{-}many\}$ |
| $A_c:\{Attr\}$ | set of constituent attributes (if respective attribute is "composite"). |

Figure 71 - DRAC Attribute Representation

Data attributes are also characterized by a cardinality, which describes the quantity of attribute instances found in the domain. Attribute cardinality is

described by one of the following values: (1) “one” - only one instance of the attribute exists; (2) “zero-to-one” - no instances exist or one instance exists; (3) “zero-to-many” - zero or more instances exist; or (4) “one-to-many” - at least one instance exists.

Each service $s \in d.S_d$ specified in the DRAC D-M is described by the structure, $(n, \theta_s, \delta, f, D_i, D_o, E_i, E_o, C_{Pre}, C_{Post})$, in Figure 72. It should be noted that functionality in the domain is typically referred to as domain tasks, where functionality in the DRA is referred to as services. To fully characterize a domain function and its input, output, and timing constraints, the service specification includes average duration and frequency as well as input data, output data, input events, output events, and pre-/post- conditions. Average duration, $s.\delta$, specifies the typical time required for service execution and is described by a value and appropriate time units (e.g., days, hours, minutes, seconds). The service frequency, $f$, describes the frequency of execution after the service’s pre-conditions have been satisfied. “Periodic” execution specifies that the service continues to be executed once per time period until the post-conditions are satisfied. “Continuous” execution is a special case of “periodic” where the service recommences immediately following termination. “Discrete” indicates that the service is executed only once.

$s : Svc \stackrel{def}{=}$

$(n, \theta_s, \delta, f, D_i, D_o, E_i, E_o, C_{Pre}, C_{Post}) \in d.S_d$

| | |
|---|---|
| $n : String$ | name of the service. |
| $\theta_s : Trc$ | service traceability. Services mirror domain tasks. |
| $\delta$ | average service duration (value and units).<br>$\delta = (\tau : \mathbb{R}, \{mins, secs, hrs, days\})$ |
| $f$ | service frequency, specified as "continuous," "discrete," or "periodic," where "continuous" is a special case of "periodic" with no interval. If "periodic," the rate is specified.<br>$f = (\{continuous, discrete, periodic\}, \tau : \mathbb{R}, \{min, sec, hr, day\})$ |
| $D_i : \{SvcInData\}$ | input data |
| $D_o : \{SvcOutData\}$ | output data |
| $E_i : \{SvcInEvt\}$ | input events |
| $E_o : \{SvcOutEvt\}$ | output events |
| $C_{Pre} : SvcPreCond$ | first order logic expression describing pre-conditions on the execution of this service. |
| $C_{Post} : SvcPostCond$ | first order logic expression describing post-conditions for this service. |

Figure 72 - DRAC Service Representation

The service pre-conditions, $s.C_{Pre}$, are first order logic expressions that specify the conditions necessary to initiate execution of the service, including the required input data and events. Similarly, the service post-conditions, $s.C_{Post}$, are first order logic expressions that describe the state of the domain after service execution has completed, including the output data produced and the events generated.

***Service input data, output data, input events, and output events***

Figure 73 - Figure 76 describe the DRA elements used to specify service input data, output data, input events, and output events, respectively. Each is identified by a name ($s.d_i.n$, $s.d_o.n$, $s.e_i.n$, and $s.e_o.n$) that uniquely identifies it among other inputs and outputs within a service, and each is traceable to a corresponding domain task input or output under the domain task to which its respective service is traced ($\theta_{di}$, $\theta_{do}$, $\theta_{ei}$, and $\theta_{eo}$). The fields $s.d_i.a$, $s.d_o.a$, $s.e_i.a$, and $s.e_o.a$ reference the respective input or output data or event for a service, where these data and events are defined as attribute in the Declarative Model of the same or another DRAC.

Input data and input event elements include a field to specify the source of the data or event ($s.d_i.sndr$ and $s.e_i.sndr$). In the large majority of cases, task inputs in the domain are described as being provided by the outputs from other tasks, thus the `sndr` field typically refers to another service in the DRA (i.e., `d.sndr:Svc`). However, it is possible that the source of the data or event is (1) external to the domain being modeled; (2) only specified as a particular domain performer (e.g., "Author") but not a specific task under that performer; or

(3) simply not specified at all. Since the DRA representation must accommodate these situations, the `sndr` field can hold a number of possible values. In addition to referring to a DRAC service, the `sndr` can be a DRAC reference (with no indication of a specific service under the DRAC), an attribute owned by some DRAC in the DRA, or the constant "external," indicating that the input is provided from outside the DRA scope. The destinations for output data and output events work similarly, where the `s.rcvr` field can refer to (1) a service in the same or another DRAC; (2) a DRAC attribute; (3) a DRAC; or (4) an external consumer.

Input and output data elements also allow the specification of cardinality, $s.d_i.k$ and $s.d_o.k$, intended to identify the quantity of the data attribute $s.d_i.a$ or $s.d_o.a$ received or sent. Cardinality can be specified as (1) one instance of the data attribute; (2) zero or one instances; (3) zero to many instances; or (4) one or more instances. In addition, output data and events provide a field for indicating the frequency, $s.d_o.f$ and $s.e_o.f$, that the data or event attribute is produced by the respective service. For example, a service may periodically update a data value during execution. Output frequency is specified in a similar manner to service execution frequency, allowing for "discrete" (i.e., data/event output once), "continuous," and "periodic" at some rate.

$d_i$:*SvcInData* $\stackrel{def}{=}$ (n,$\theta_{di}$,a,k,sndr) $\in$ s.$D_i$

| | |
|---|---|
| n:*String* | name of the data input (referenced in preconditions). |
| $\theta_{di}$:*Trc* | input data traceability. Service input data is supported by corresponding task input data. |
| a:*Attr* | data attribute input. Data *must* be defined in some DRAC but can be *any* DRAC.<br>$a \in \{a_1$:*Attr* $\mid a_1.y \neq$ Event $\wedge$<br>$(\exists d_1$:*Drac* $\mid a_1 \in d_1.A_d \wedge$<br>refArch($d_1$) = refArch($d_i$))$\}$ |
| k | cardinality (quantity) of data received.<br>k $\in$ {one, zero-to-one,<br>zero-to-many, one-to-many} |
| sndr | source of input data. Can be specified as another service (in any DRAC), an attribute (in any DRAC), or "external". If source is from an attribute, that attribute must be the same as the attribute specified in $d_i$.a above.<br>sndr $\in \{a_1$:*Attr* $\mid a_1 = d_i.a\} \cup$<br>$\{s_1$:*Svc* $\mid (\exists d_1$:*Drac* $\mid s_1 \in d_1.S_d) \wedge$<br>refArch($d_1$) =<br>refArch($d_i$)$\} \cup$ {External} |

Figure 73 - DRAC Service Input Data Representation

| $d_o$:*SvcOutData* $\stackrel{def}{=}$(n,$\theta_{do}$,a,k,f,rcvr) $\in$ s.$D_o$ | |
|---|---|
| n:*String* | name of the data output (referenced in postconditions). |
| $\theta_{do}$:*Trc* | output data traceability. Service output data is supported by corresponding task output data |
| a:*Attr* | data attribute input. Data *must* be defined in some DRAC but can be *any* DRAC.<br>$a \in \{a_1:Attr \mid a_1.y \neq Event \wedge (\exists d_1:Drac \mid a_1 \in d_1.A_d \wedge refArch(d_1) = refArch(d_o))$ |
| k | cardinality (quantity) of data sent.<br>k $\in$ {one,zero-to-one, zero-to-many,one-to-many} |
| f | frequency this data is output by this service (represented similar to "service frequency" above). |
| rcvr | destination of output data. Can be specified as another service (in any DRAC), an attribute (in any DRAC), or "external". If destination is to an attribute, that attribute must be the same as the attribute specified in $d_o$.a above.<br>$rcvr \in \{a_1:Attr \mid a_1 = d_o.a\} \cup \{s_1:Svc \mid (\exists d_1:Drac \mid s_1 \in d_1.S_d) \wedge refArch(d_1) = refArch(d_o)\} \cup \{External\}$ |

Figure 74 - DRAC Service Output Data Representation

| $e_i$:*SvcInEvt* $\triangleq$ ($n, \theta_{ei}, a, sndr$) $\in$ $s.E_i$ | |
|---|---|
| $n$:*String* | name of the event input (referenced in preconditions). |
| $\theta_{ei}$:*Trc* | input event traceability. Service input event is supported by corresponding task input event. |
| $a$:*Attr* | event attribute input. Event *must* be defined in some DRAC but can be *any* DRAC.<br>$a \in \{a_1$:*Attr* $\mid a_1.y = Event \wedge$<br>$(\exists d_1$:*Drac* $\mid a_1 \in d_1.A_d \wedge$<br>$refArch(d_1) = refArch(e_i))\}$ |
| sndr | source of input event. Can be specified as another service (in any DRAC), an attribute (in any DRAC), or "external". If source is from an attribute, that attribute must be the same as the attribute specified in $e_i.a$ above.<br>$sndr \in \{a_1$:*Attr* $\mid a_1 = e_i.a\} \cup$<br>$\{s_1$:*Svc* $\mid (\exists d_1$:*Drac* $\mid$<br>$s_1 \in d_1.S_d) \wedge$<br>$refArch(d_1) =$<br>$refArch(e_i)\} \cup \{External\}$ |

Figure 75 - DRAC Service Input Event Representation

| $e_o$:*SvcOutEvt* $\triangleq$ (n, $\theta_{eo}$, a, f, rcvr) $\in$ $s.E_o$ | |
|---|---|
| n:*String* | name of the event input (referenced in postconditions). |
| $\theta_{eo}$:*Trc* | output event traceability. Service output event is supported by corresponding task output event. |
| a:*Attr* | event attribute output. Event *must* be defined in some DRAC but can be *any* DRAC.<br>$a \in \{a_1$:*Attr* $\mid a_1.y$ = Event $\wedge$<br>$(\exists d_1$:*Drac* $\mid a_1 \in d_1.A_d \wedge$<br>refArch($d_1$) = refArch($d_i$))} |
| f | frequency this event is output by this service (represented similar to "service frequency" above). |
| rcvr | destination of output event. Can be specified as another service (in any DRAC), an attribute (in any DRAC), or "external". If destination is to an attribute, that attribute must be the same as the attribute specified in $e_o$.a above.<br>rcvr $\in \{a_1$:*Attr* $\mid a_1 = e_o.a\} \cup$<br>$\{s_1$:*Svc* $\mid (\exists d_1$:*Drac* $\mid$<br>$s_1 \in d_1.S_d) \wedge$<br>refArch($d_1$) =<br>refArch($e_o$) $\} \cup \{$External$\}$ |

Figure 76 - DRAC Service Output Event Representation

***Service pre- and post- conditions***

The DRA is designed to capture domain requirements, described by domain data concepts, domain processes, and the timing between those processes (e.g., order of execution). Domain processes and data are described by DRAC

services and related data attributes. Timing is described by pre- and post-conditions that express the conditions that must exist prior to service initiation and those conditions that will be present as a result of service completion. Timing specifications include not only input/output data and events but also required service execution sequence (i.e., task1 must complete before task2 commences) and the necessary state of data elements given by the data element value (e.g., data attribute $d_1.attr_1$ must be greater than 5). Figure 77 shows the predicates used to express these condition.

The Declarative Model identifies the domain data, events, and functionality allocated to a respective DRAC from the Domain Model, thus specifying the set of domain requirements that a technology instantiating the DRAC must provide. The specification is intentionally implementation independent to accommodate all forms of technology, including software, hardware, and personnel. Attributes, services, and input/output data and events in the Declarative Model are explicitly traceable to corresponding data, events, tasks, and inputs and outputs in the Domain Model.

| $C_{Pre}$:*SvcPreCond*, $C_{Post}$:*SvcPostCond* |
|---|
| Service pre-conditions and post-conditions are first order logic expressions utilizing one or more of the following predicates:<br><br>Data value comparison (i.e., particular data value is equal to, less than, greater than some other value), where the value of a data attribute is referenced using `DATA_VALUE( <DRAC name>,<attribute name>)`. For example, if a condition required attribute "A1" in DRAC "D1" to be greater than 5, the pre-/post- condition would be expressed as `DATA_VALUE("D1","A1") > 5`.<br><br>Availability of service input data, expressed as `INPUT_DATA_AVAILABLE(<unique service input data name>)`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.<br><br>Presence of triggering service input event, expressed as `INPUT_EVENT_GENERATED(<unique service input event name>)`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.<br><br>Production of service output data, expressed as `OUTPUT_DATA_AVAILABLE(<unique service output data name>)`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.<br><br>Generation of service output event, expressed as `OUTPUT_EVENT_GENERATED(<unique service output event name>)`, where the respective service is considered to be the one for which the pre- or post- condition is being defined.<br><br>Completion of another service, expressed as `AFTER_TERMINATION_OF(<DRAC name> , <service name> )` |

Figure 77 - DRAC Service Pre-/Post- Condition Representation

**DRAC Behavioral Model (DRAC B-M) Representation**

The DRAC B-M contains a high-level state chart representing DRAC behavior. This behavior reflects service execution and the changes in state that result from that execution. Describing behavior using a state chart representation makes it possible to leverage existing techniques for evaluating prescribed system behavior with respect to completeness and correctness properties, including safety and liveness.

Since the DRA representation emphasizes implementation independence, the states in the DRAC B-M and the transitions between those states are described by referencing elements in the DRA, rather than the detailed states and transitions corresponding to a particular technology.

A transition between states, $t \in sc.T_{sc}$, is described using the five-tuple, $(\theta_t, from, to, C_{guard}, \delta_t)$, shown in Figure 78. A transition specification includes the source and destination states, $t.from$ and $t.to$, and a guard, $t.C_{guard}$, that describes the conditions that must be true for the transition to be enabled. Guards are specified as first order logic expressions:

- DATA_VALUE,
- AFTER_TERMINATION_OF,
- INPUT_EVENT_GENERATED,
- INPUT_DATA_AVAILABLE,
- OUTPUT_EVENT_GENERATED, and
- OUTPUT_DATA_AVAILABLE.

$sc:StChart \stackrel{def}{=} (S_{sc}, T_{sc})$

$\theta_{sc}:Trc$ Statechart traceability.

$S_{sc}:\{State\}$ set of states in the statechart. Each state may also be decomposed into a lower-level statechart.

$s:State \stackrel{def}{=} (n, \theta_s, sc_{sub}, Q, \delta_s) \in sc.S_{sc}$

- $n:String$ state name
- $\theta_s:Trc$ state traceability
- $sc_{sub}:StChart$ lower-level statechart describing substates within this state $s$.
- $Q$ state qualifiers

  $Q \subseteq \{start\text{-}state, exit\text{-}safe\}$
- $\delta_s$ average state duration (value and units).

  $\delta = (\tau:\mathbb{R}, \{mins, secs, hrs, days\})$

$T_{sc}:\{Transition\}$ et of transitions between states in the statechart.

$t:Transition \stackrel{def}{=} (\theta_t, from, to, C_{guard}, \delta_t) \in sc.T_{sc}$

- $\theta_t:Trc$ transition traceability
- $from:State$ transition source state
- $to:State$ transition destination state
- $C_{guard}:Guard$ first order logic expression describing when transition is enabled.
- $\delta_t$ average transition duration (value and units).

  $\delta = (\tau:\mathbb{R}, \{mins, secs, hrs, days\})$

Figure 78 - DRA Satechart Representation

Since a DRAC's B-M often reflects content in DRAC D-M services (i.e., service-based approach), predicates used for service pre- and post- conditions are closely related to those available to describe transition guards. An average duration can be specified for both states and transitions ($s.\delta_s$, $t.\delta_t$).

**DRAC Integration Model (DRAC I-M) Representation**

The DRAC Integration Model (DRAC I-M) describes (1) dependencies between DRACs, (2) dependencies between services in separate DRACs, and (3) dependencies between a service and an attribute, where each is owned by a separate DRAC. Dependency knowledge provided by the DRAC I-M is important when attempting to determine the degree of coupling between DRACs, services, and attributes. Dependencies between DRACs, services, and attributes arise for the following reasons:

1) Service-to-Service Dependencies: A service provided by one DRAC requires input data or events produced by another service located in another DRAC. For example, "Service 1" in "DRAC 1" is coupled to "Service 2" in "DRAC 2" because "Service 1" requires data "Data 1" produced by "Service 2."

2) Service-to-Attribute Dependencies: A service provided by one DRAC requires an input data or event that it receives directly from an attribute located in another DRAC (an attribute not represented as being generated from another service found in the DRA). "Service 1" in

"DRAC 1" is coupled to "Attribute 1" in "DRAC 3" because "Service 1" requires "Event 1" as input, which is owned by "DRAC3" as "Attribute 1."

3) Subsystem DRAC Dependencies: Two or more DRACs have been collected into a subsystem definition. DRACs "DRAC 1," "DRAC 4," "DRAC5," and "DRAC6" are coupled because they all belong to subsystem "Subsystem 1."

# APPENDIX B - DRA TO ARCADE METAMODEL ALGORITHM

The following algorithm is used to convert a SEPA Domain Reference Architecture (DRA) into an Arcade Architecture. The DRA metamodel does not contain information regarding compute environments and communication channels, therefore these portions of the Arcade metamodel are not populated from DRAs. The algorithm uses the DRA metamodel defined in APPENDIX A.

```
Algorithm translateReferenceArch(ra : RefArch) : ARCH

BEGIN // main algorithm
 arch = new ARCH
 arch.n = ra.n
 // for each DRAC in ra make an Arcade Component
 ∀ d: DRAC ∈ ra.Dra
      arch.C = arch.C ∪ newComponent(d)
 // make the Arcade connectors
 arch.Q = makeConnectors(ra,arch)
 return arch
END // main algorithm
```

newComponent(d: *DRAC*) : *Component*

// create and populate an Arcade component

// with attributes and services

c = new *Component*

c.n = d.n

∀ a: *Attr* ∈ $d.A_d$

c.A = c.A ∪ newAttribute(a)

∀ s: *Svc* ∈ $d.S_d$

c.S = c.S ∪ newService(s)

return c

newAttribute(a: *Attr*) : *Attribute*

// create an Arcade attribute from the RA Attr

at = new *Attribute*

at.n = a.n

at.t = a.y

$at.A = a.A_c$

return at

newService(s: *Svc*) : *Service*

// create an Arcade Service from the DRA Svc

srv = new *Service*

srv.n = s.n

srv.δ = s.δ

// map all the in & out data & events

∀ $d_i$: *SvcInData* ∈ s.$D_i$

srv.$D_i$ = srv.$D_i$ ∪ newInDataParameter($d_i$)

∀ $d_o$: *SvcOutData* ∈ s.$D_o$

srv.$D_o$ = srv.$D_o$ ∪ newOutDataParameter($d_o$)

∀ $e_i$: *SvcInEvent* ∈ s.$E_i$

srv.$E_i$ = srv.$E_i$ ∪ newInEventParameter($e_i$)

∀ $e_o$: *SvcOutEvent* ∈ s.$E_o$

srv.$E_o$ = srv.$E_o$ ∪ newOutEventParameter($e_o$)

// map the pre- and post-conditions

srv.$C_{pre}$ = s.$C_{pre}$

srv.$C_{post}$ = s.$C_{post}$

return srv

newInDataParameter(d: *SvcInData*) : *Parameter*

// map an input data to an Arcade Parameter

p = new *Parameter*

p.n = d.n

p.k = inData

return p

```
newOutDataParameter(d: SvcOutData) : Parameter
// map an output data to an Arcade Parameter
p = new Parameter
p.n = d.n
p.k = outData
return p


newInEventParameter(d: SvcInEvent) : Parameter
// map an input event to an Arcade Parameter
p = new Parameter
p.n = d.n
p.k = inEvent
return p


newOutEventParameter(d: SvcOutEvent) : Parameter
// map an output event to an Arcade Parameter
p = new Parameter
p.n = d.n
p.k = outEvent
return p
```

```
makeConnectors(ra: RefArch, arch: ARCH) : {Connector)
C : {Connector} = {}

// create all Internal data to data connectors
∀ d1,d2: DRAC ∈ ra.dra | d1 ≠ d2
  ∀ s1: Service ∈ d1.S, s2: Service ∈ d2.S | s1 ≠ s2
      ∀ p1: SvcOutData ∈ s1.Do, p2: SvcInData ∈ s2.Di
            | p1.rcvr = s2 ^ p2.sndr = s1
        C = C ∪ newConnector(parmFor(p1), parmFor(p2),
                             p1.a, p2.a)

// create all external to internal data connectors
∀ d: DRAC ∈ ra.dra
  ∀ s1: Service ∈ d.S
      ∀ p1: SvcInData ∈ s1.Di | p1.sndr = External
        C = C ∪ newConnector(External, parmFor(p1),
                             NULL, P1.a)

// create all internal to external data connectors
∀ d: DRAC ∈ ra.dra
  ∀ s1: Service ∈ d.S
      ∀ p1: SvcOutData ∈ s1.Do | p1.rcvr = External
        C = C ∪ newConnector(parmFor(p1), External,
                             P1.a, NULL)
```

(continued on next page)

makeConnectors(continued from previous page)

```
// create all internal event to event connectors
∀ d1,d2: DRAC ∈ ra.dra | d1 ≠ d2
  ∀ s1: Service ∈ d1.S, s2: Service ∈ d2.S | s1 ≠ s2
      ∀ p1: SvcOutEvent ∈ s1.Do, p2: SvcInEvent ∈ s2.Di
            | p1.rcvr = s2 ^ p2.sndr = s1
        C = C ∪ newConnector(parmFor(p1), parmFor(p2),
                             p1.a, p2.a)

// create all external to internal event connectors
∀ d: DRAC ∈ ra.dra
  ∀ s1: Service ∈ d.S
      ∀ p1: SvcInEvent ∈ s1.Di | p1.sndr = External
        C = C ∪ newConnector(External, parmFor(p1),
                             NULL, P1.a)

// create all internal to external event connectors
∀ d: DRAC ∈ ra.dra
  ∀ s1: Service ∈ d.S
      ∀ p1: SvcOutEvent ∈ s1.Do | p1.rcvr = External
        C = C ∪ newConnector(parmFor(p1), External,
                             P1.a, NULL)

// create all attribute to inData connectors
∀ d: DRAC ∈ ra.dra
  ∀ s1: Service ∈ d.S
      ∀ p1: SvcInData ∈ s1.Di | p1.sndr = Attribute
        C = C ∪ newConnector(NULL, parmFor(p1),
                             NULL, P1.a)
```

(continued on next page)

makeConnectors(continued from previous page)

```
// create all attribute to inEvent connectors
∀ d: DRAC ∈ ra.dra
  ∀ s1: Service ∈ d.S
      ∀ p1: SvcInEvent ∈ s1.Di | p1.sndr = Attribute
        C = C ∪ newConnector(NULL, parmFor(p1),
                             NULL, P1.a)

// create all attribute to outData connectors
∀ d: DRAC ∈ ra.dra
  ∀ s1: Service ∈ d.S
      ∀ p1: SvcOutData ∈ s1.Do | p1.rcvr = Attribute
        C = C ∪ newConnector(parmFor(p1), NULL,
                             P1.a, NULL)

// create all attribute to outEvent connectors
∀ s1: Service ∈ d.S
  ∀ p1: SvcOutEvent ∈ s1.Do | p1.rcvr = Attribute
        C = C ∪ newConnector(parmFor(p1), NULL,
                             P1.a, NULL)\

return C
```

parmFor(p: {SvcInData,SvcOutData,SvcInEvent,SvcOutEvent}) : Parameter

This function is defined as finding and returning the *Parameter* that has been created for a given instance of {SvcInData, SvcOutData, SvcInEvent, SvcOutEvent}

# APPENDIX C - ARCADE TO PROMELA ALGORITHM

The algorithm for generating the Promela code for an Arcade Architecture model is defined below. This algorithm uses the Arcade metamodel definitions in Section 3.2.2.

```
Algorithm genPromela(a : ARCH)
BEGIN // main algorithm
 // for each Component in ARCH generate the data structures
 ∀ c: Component ∈ a.C
      genDataStructs(c)
 // for each Component in ARCH generate the Promela
 // channels that reflect the connectors
 ∀ x: Connector ∈ a.X
      genChannelDefinition(x)
 // for each Component in ARCH generate the Promela process
 // and generate the state logic for pre/post conditions
 ∀ c: Component ∈ a.C
      genStateLogic(c)
 // generate the code to manage exchange of external data
 // and events required by ARCH (e.g., close the model)
 genExternalModel(a)
 // generate the code that will bootstrap scenarios by
 // sending initial data and events
 genScenarios(a)
 // generate Promela init process to connect all channels,
 // and start all other processes
 genInit(a)
END // main algorithm
```

```
genDataStructs(c: Component)
 // generate the data structures for this component
 ∀ a: Attr ∈ c.A_c
      genAttributeDataStruct(a)
 ∀ s: Svc ∈ c.S_c
      genServiceDataStruct(s)
```

```
genStateLogic(c: Component)
 // generate the Promela process for this component, and
 // generate the state logic for pre/post-conditions
 ∀ s: Svc ∈ c.S_c
      genPreConditionCode(s.C_pre)
      genServiceExecutionCode(s)
      genPostConditionCode(s.C_post)
```

```
genExternalModel(a : ARCH)
 // generate the code to manage exchange of external data
 // and events required by the ARCH (e.g., close the model)
 ∀ c: Component ∈ a.C
    ∀ s: Svc ∈ c.S_c | (hasExternalInputs(s) ^
                        hasInternalInputs(s))
      genExternalInputSource(s)
```

```
genScenarios(a : ARCH)
 // generate the code to bootstrap scenarios by sending
 // initial data and events; this is done for all services
 // that have only 'External' inputs
 ∀ c: Component ∈ a.C
      ∀ s: Svc ∈ c.S_c |
           (hasExternalInputs(s) ^ ¬hasInternalInputs(s))
      genScenarioInputSource(s)


genInit(a : ARCH)
 // generate the code to assign both
 // ends of channels to the correct processes
 ∀ x: Connector ∈ a.X
    genChanAssignment(x)

 // generate the code to start the component processes, and
 // the External & Scenarios processes
 ∀ c: Component ∈ a.C
      genProcessInit(c)

 genExternalInit()
 genScenariosInit()
```

# APPENDIX D - DETAILED EXPERIMENTAL RESULTS

## EXPERIMENT 1: DYNAMIC PROPERTY DEPENDENCIES

Details of the experimental setup and conclusions for this experiment are presented in Section 5.6.1.

The Spearman coefficients that were calculated for each pair of dynamic properties are shown in Table 37. The associated *p-values* are shown in Table 38. The confidence levels associated with rejecting $H_0$ for correlations with significant *p-values* are shown in Table 39 (property pairs with no entries in this table showed no correlation according to the Spearman test; entries for pairs with correlations indicate the confidence level of the Spearman coefficient). The $R^2$ values for significant correlations are summarized by property in Table 40 - Table 50, along with the final determination of the goodness-of-fit and types of correlations (e.g., positive or negative) that were detected using them.

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SAFETY | 1.00 | | | | | | | | | | |
| LIVENESS | 0.93 | 1.00 | | | | | | | | | |
| USAGE PROFILE LATENCY | 0.45 | 0.45 | 1.00 | | | | | | | | |
| USAGE PROFILE THROUGHPUT | 0.27 | 0.37 | -0.34 | 1.00 | | | | | | | |
| COMPONENT UTILIZATION | -0.12 | -0.15 | 0.38 | -0.24 | 1.00 | | | | | | |
| COMPONENT THROUGHPUT | -0.38 | -0.42 | 0.26 | -0.59 | 0.75 | 1.00 | | | | | |
| SERVICE LATENCY | 0.20 | 0.08 | 0.81 | -0.62 | 0.57 | 0.56 | 1.00 | | | | |
| SERVICE UTILIZATION | 0.06 | -0.12 | -0.35 | 0.21 | 0.03 | -0.23 | -0.04 | 1.00 | | | |
| SERVICE THROUGHPUT | -0.33 | -0.46 | -0.78 | 0.26 | -0.20 | -0.11 | -0.46 | 0.64 | 1.00 | | |
| COMPONENT RELIABILITY | -0.33 | -0.35 | -0.12 | -0.08 | 0.37 | 0.23 | 0.25 | 0.58 | 0.34 | 1.00 | |
| SERVICE RELIABILITY | -0.82 | -0.64 | -0.25 | -0.18 | 0.24 | 0.25 | -0.11 | -0.07 | 0.03 | 0.51 | 1.00 |

Table 37 - Spearman Correlation (ρ) Matrix for Experiment 1

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SAFETY | | | | | | | | | | | |
| LIVENESS | <.0001 | | | | | | | | | | |
| USAGE PROFILE LATENCY | 0.0065 | 0.0068 | | | | | | | | | |
| USAGE PROFILE THROUGHPUT | 0.1201 | 0.0267 | 0.0462 | | | | | | | | |
| COMPONENT UTILIZATION | 0.4824 | 0.3761 | 0.0246 | 0.1608 | | | | | | | |
| COMPONENT THROUGHPUT | 0.0236 | 0.0131 | 0.1359 | 0.0002 | <.0001 | | | | | | |
| SERVICE LATENCY | 0.2585 | 0.6501 | <.0001 | <.0001 | 0.0004 | 0.0005 | | | | | |
| SERVICE UTILIZATION | 0.7292 | 0.5009 | 0.0389 | 0.2258 | 0.8480 | 0.1833 | 0.8118 | | | | |
| SERVICE THROUGHPUT | 0.0509 | 0.0049 | <.0001 | 0.1320 | 0.2440 | 0.5375 | 0.0054 | <.0001 | | | |
| COMPONENT RELIABILITY | 0.0533 | 0.0414 | 0.4979 | 0.6623 | 0.0289 | 0.1913 | 0.1412 | 0.0003 | 0.0838 | | |
| SERVICE RELIABILITY | <.0001 | <.0001 | 0.1420 | 0.2947 | 0.1697 | 0.1491 | 0.5138 | 0.6939 | 0.8731 | 0.0018 | |

Table 38 - Two-tailed *p-values* for Experiment 1

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SAFETY | | 0.99 | 0.99 | | | 0.95 | | | | | 0.99 |
| LIVENESS | 0.99 | | 0.99 | 0.95 | | 0.98 | | | 0.99 | 0.95 | 0.99 |
| USAGE PROFILE LATENCY | 0.99 | 0.99 | | 0.95 | 0.95 | | 0.99 | 0.95 | 0.99 | | |
| USAGE PROFILE THROUGHPUT | | 0.95 | 0.95 | | | 0.99 | 0.99 | | | | |
| COMPONENT UTILIZATION | | | 0.95 | | | 0.99 | 0.99 | | | 0.95 | |
| COMPONENT THROUGHPUT | 0.95 | 0.98 | | 0.99 | 0.99 | | 0.99 | | | | |
| SERVICE LATENCY | | | 0.99 | 0.99 | 0.99 | 0.99 | | | 0.99 | | |
| SERVICE UTILIZATION | | | 0.95 | | | | | | 0.99 | 0.99 | |
| SERVICE THROUGHPUT | | 0.99 | 0.99 | | | | 0.99 | 0.99 | | | |
| COMPONENT RELIABILITY | | 0.95 | | | 0.95 | | | 0.99 | | | 0.99 |
| SERVICE RELIABILITY | 0.99 | 0.99 | | | | | | | | 0.99 | |

Table 39 - Confidence Levels for Rejecting $H_0$ for Experiment 1

| | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | 0.99 | 0.99 | | | 0.95 | | | | | 0.99 |
| Regression $R^2$ | 0.94 | 0.20 | | | 0.06 | | | | | 0.75 |
| Correlation | Strong (+) | NONE | NONE | NONE | NONE | NONE | NONE | NONE | NONE | Moderate (-) |

Table 40 - Safety Correlation Results

| | SAFETY | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | 0.99 | | 0.95 | | 0.98 | | | 0.99 | 0.95 | 0.99 |
| Regression $R^2$ | 0.94 | | 0.10 | | 0.07 | | | 0.17 | 0.19 | 0.59 |
| Correlation | Strong (+) | NONE | NONE | NONE | NONE | NONE | NONE | NONE | NONE | Weak (-) |

Table 41 - Liveness Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | 0.99 | 0.99 | 0.95 | 0.95 | | 0.99 | 0.95 | 0.99 | | |
| Regression $R^2$ | 0.20 | 0.23 | 0.35 | 0.10 | | 0.76 | 0.28 | 0.77 | | |
| Correlation | NONE | NONE | NONE | NONE | NONE | Moderate (+) | NONE | Moderate (-) | NONE | NONE |

Table 42 - Usage Profile Latency Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | | 0.95 | 0.95 | | 0.99 | 0.99 | | | | |
| Regression $R^2$ | | 0.10 | 0.35 | | 0.50 | 0.46 | | | | |
| Correlation | NONE | NONE | NONE | NONE | Moderate (-) | NONE | NONE | NONE | NONE | NONE |

Table 43 - Usage Profile Throughput Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | | | 0.95 | | 0.99 | 0.99 | | | 0.95 | |
| Regression $R^2$ | | | 0.10 | | 0.77 | 0.40 | | | 0.12 | |
| Correlation | NONE | NONE | NONE | NONE | Moderate (+) | NONE | NONE | NONE | NONE | NONE |

Table 44 - Component Utilization Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | 0.95 | 0.98 | | 0.99 | 0.99 | 0.99 | | | | |
| Regression $R^2$ | 0.06 | 0.07 | | 0.50 | 0.77 | 0.43 | | | | |
| Correlation | NONE | NONE | NONE | Moderate (-) | Moderate (+) | NONE | NONE | NONE | NONE | NONE |

Table 45 - Component Throughput Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | | | 0.99 | 0.99 | 0.99 | 0.99 | | 0.99 | | |
| Regression $R^2$ | | | 0.76 | 0.46 | 0.40 | 0.43 | | 0.45 | | |
| Correlation | NONE | NONE | Moderate (+) | NONE | NONE | NONE | NONE | NONE | NONE | NONE |

Table 46 - Service Latency Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE THROUGHPUT | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | | | 0.95 | | | | | 0.99 | 0.99 | |
| Regression $R^2$ | | | 0.28 | | | | | 0.59 | 0.27 | |
| Correlation | NONE | NONE | NONE | NONE | NONE | NONE | NONE | Weak (+) | NONE | NONE |

Table 47 - Service Utilization Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | COMPONENT RELIABILITY | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | | 0.99 | 0.99 | | | | 0.99 | 0.99 | | |
| Regression $R^2$ | | 0.17 | 0.77 | | | | 0.45 | 0.59 | | |
| Correlation | NONE | NONE | Moderate (-) | NONE | NONE | NONE | NONE | Weak (+) | NONE | NONE |

Table 48 - Service Throughput Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | SERVICE RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | | 0.95 | | | 0.95 | | | 0.99 | | 0.99 |
| Regression $R^2$ | | 0.19 | | | 0.12 | | | 0.27 | | 0.16 |
| Correlation | NONE | NONE | NONE | NONE | NONE | NONE | NONE | NONE | NONE | NONE |

Table 49 - Component Reliability Correlation Results

| | SAFETY | LIVENESS | USAGE PROFILE LATENCY | USAGE PROFILE THROUGHPUT | COMPONENT UTILIZATION | COMPONENT THROUGHPUT | SERVICE LATENCY | SERVICE UTILIZATION | SERVICE THROUGHPUT | COMPONENT RELIABILITY |
|---|---|---|---|---|---|---|---|---|---|---|
| H1 Confidence | 0.99 | 0.99 | | | | | | | | 0.99 |
| Regression $R^2$ | 0.75 | 0.59 | | | | | | | | 0.16 |
| Correlation | Moderate (-) | Weak (-) | NONE | NONE | NONE | NONE | NONE | NONE | NONE | NONE |

Table 50 - Service Reliability Correlation Results

### EXPERIMENT 2: EFFECT OF STRUCTURAL DECISIONS

Details of the experimental setup and conclusions for this experiment are presented in Section 5.8.1.

The experimental results are presented as combination charts in this section. There is one chart for each property evaluated in this experiment. These charts consist of box-and-whiskers plots representing the descriptive statistics for property exceptions, and line graphs representing the number of property exceptions for each architecture. The box-and-whiskers charts are associated with the left-hand Y-axis. The top-most and bottom-most hash marks represent min and max. The box is bounded on the lower side by the 1st quartile statistic and on the upper side by the third quartile statistic. The diamond shape marks the median for property exceptions. The line graph represents the number of property exceptions, and is associated with the right-hand Y-axis. Results for all architectures are shown in ascending order left to right by number of exceptions. The intent of representing the results in this format is to show the number and types of property exceptions along with a good indication of the variance in the metric values that constituted property exceptions.

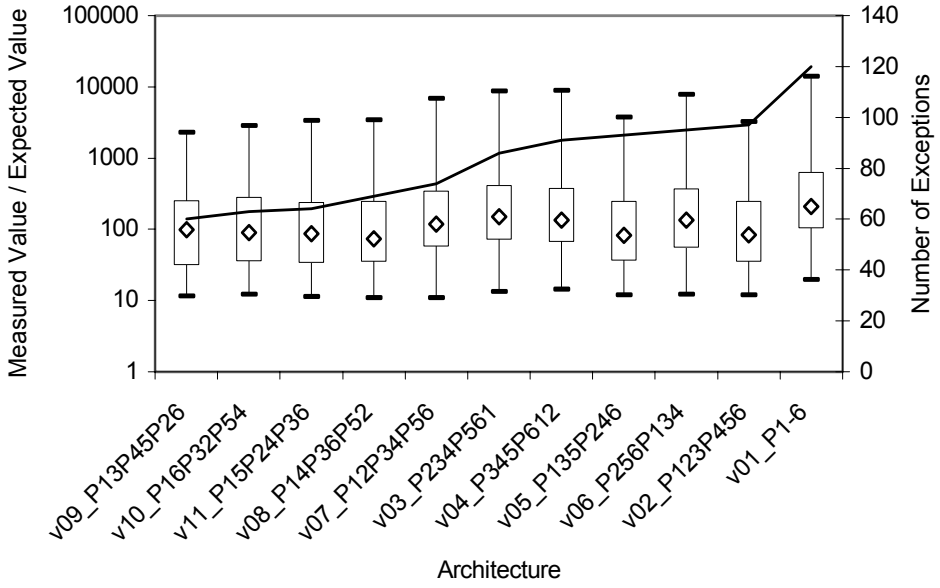


Figure 79 - Usage Profile Latency: $EXCP(L_{UP})$

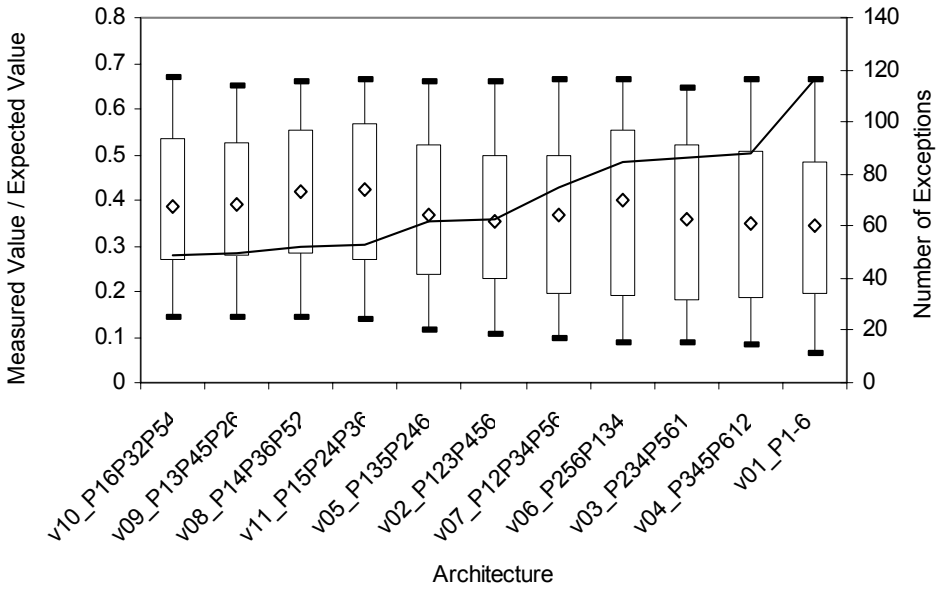


Figure 80 - Usage Profile Throughput: $EXCP(TP_{UP})$

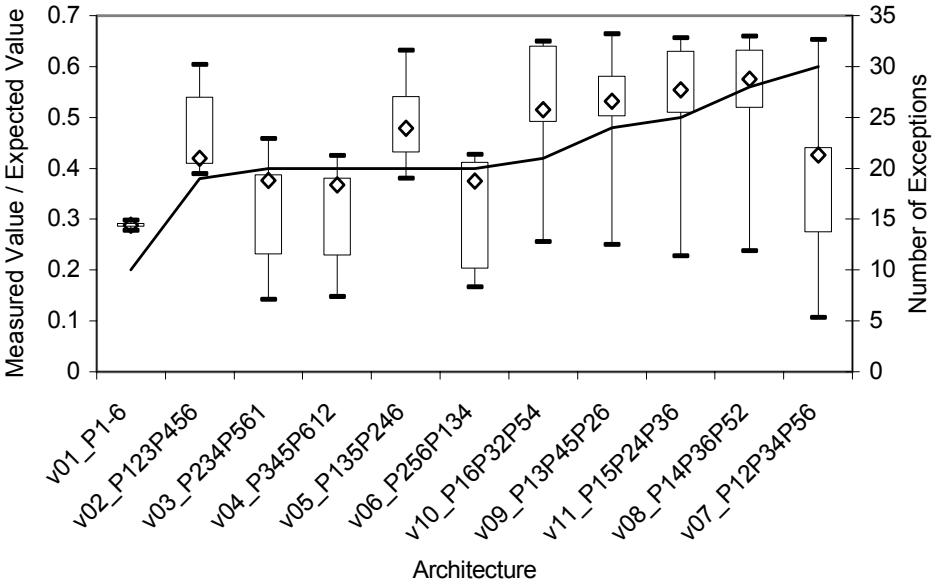


Figure 81 - Component Utilization: *EXCP*($U_C$)

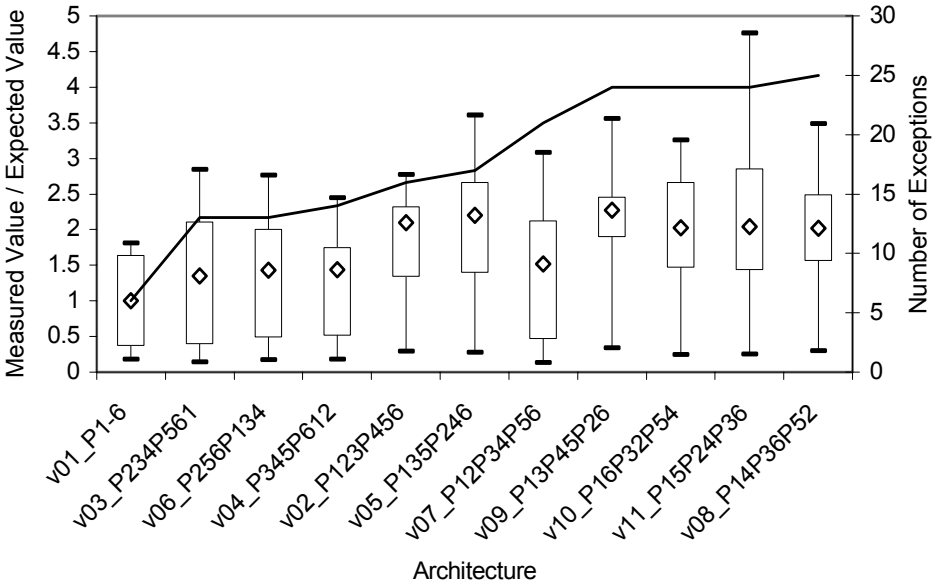


Figure 82 - Component Throughput: *EXCP*($TP_C$)

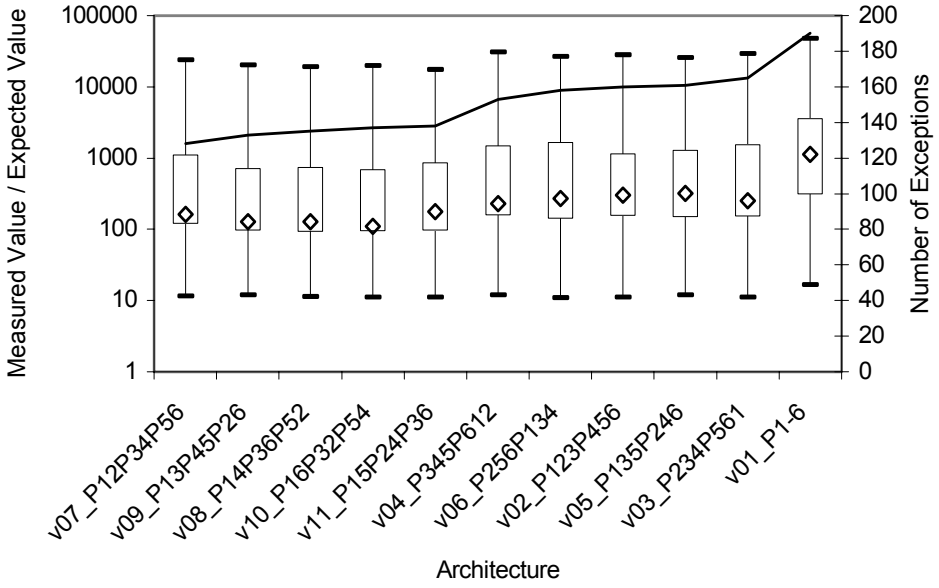


Figure 83 - Service Latency: *EXCP*($L_{SVC}$)

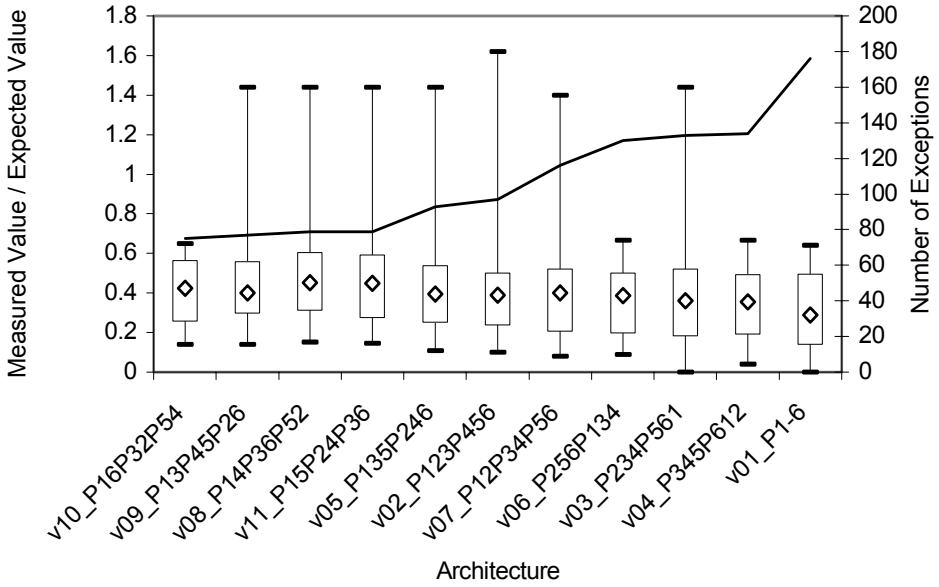


Figure 84 - Service Utilization: *EXCP*($U_{SVC}$)

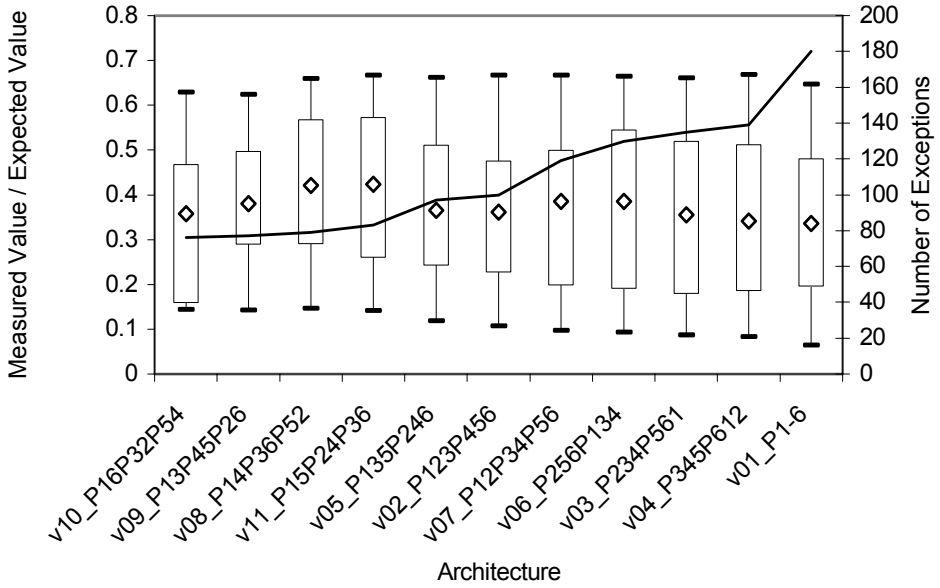


Figure 85 - Service Throughput: $EXCP(TP_{SVC})$

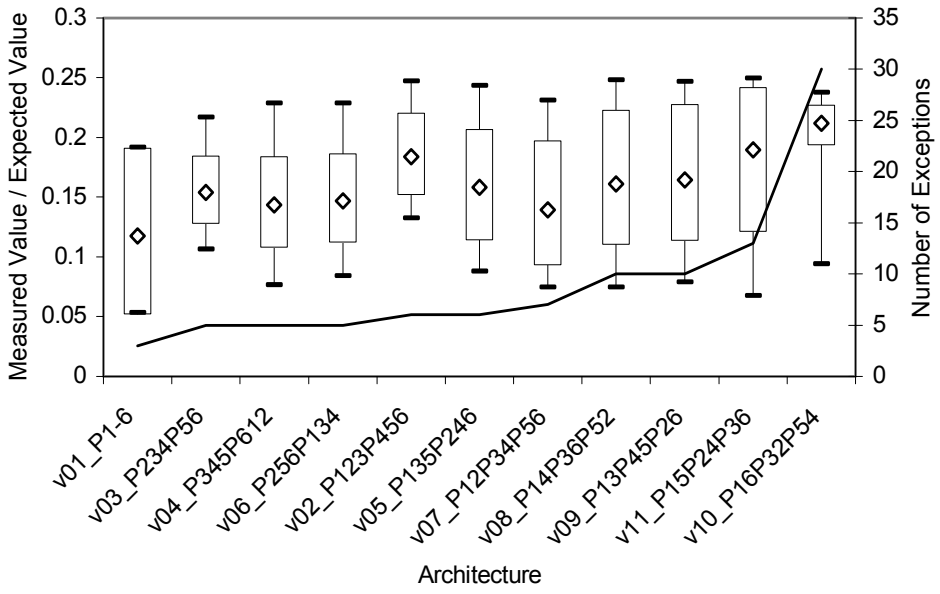


Figure 86 - Component Reliability: $EXCP(R_C)$

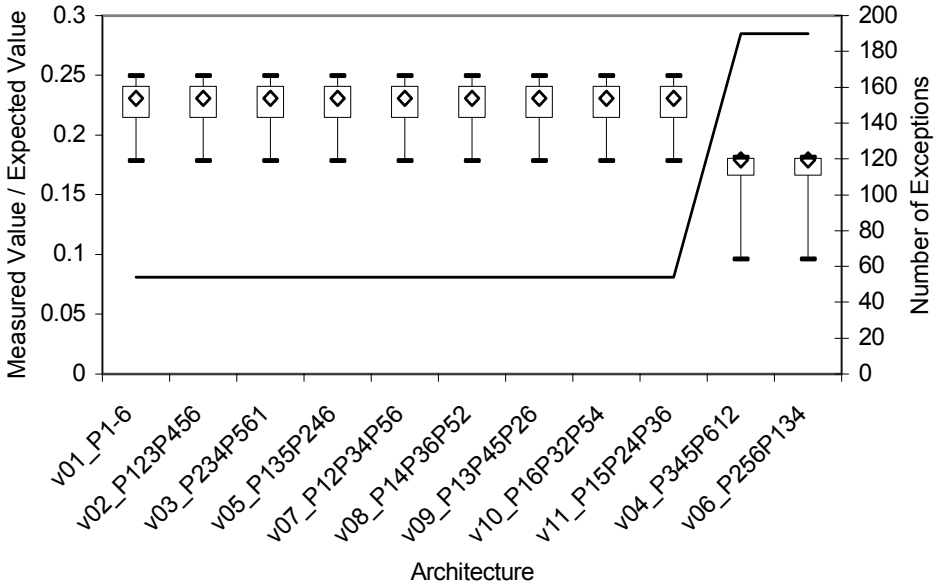


Figure 87 - Service Reliability: $EXCP(R_{SVC})$

## EXPERIMENT 3: EFFECTS OF WITHIN-ARCHITECTURE EVOLUTION

Details of the experimental setup and conclusions for this experiment are presented in Section 5.8.3.

Rankings for each of the six DRA versions at each of the seven requirements revision levels are shown in Figure 88 through Figure 94. This is followed by an analysis of correlation of the rankings of the DRA versions across requirements revision levels.

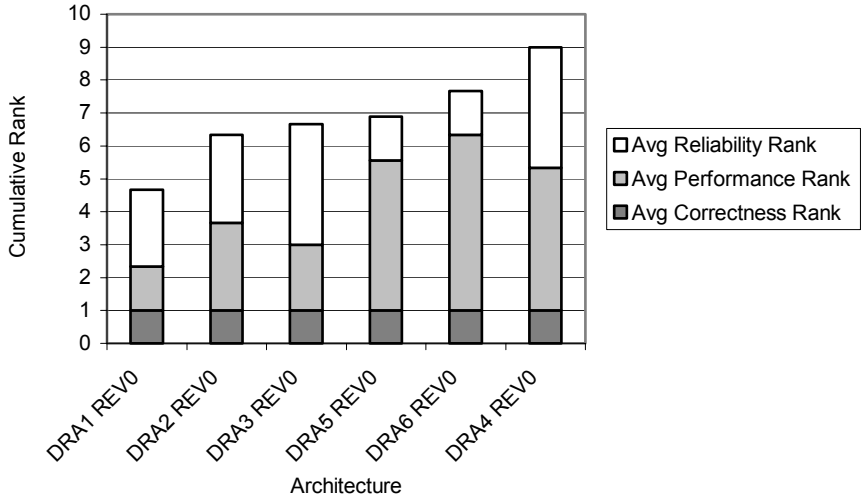


Figure 88 - Cumulative Rankings for REV0 DRAs

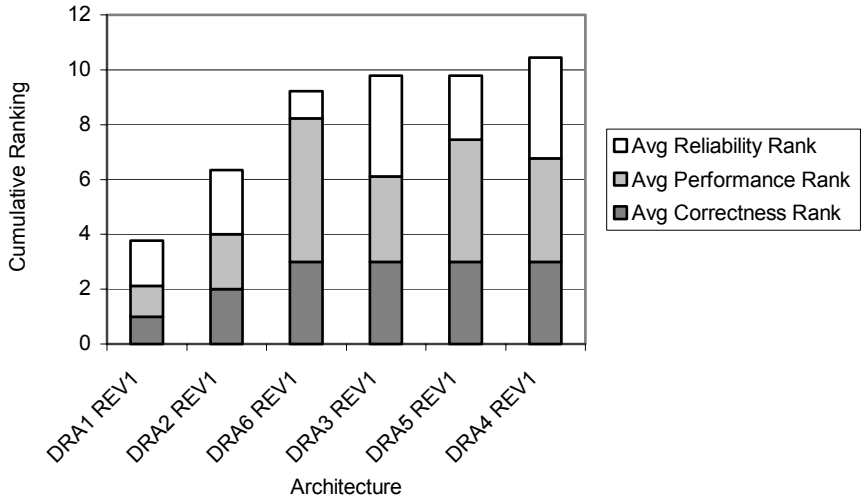


Figure 89 - Cumulative Rankings for REV1 DRAs

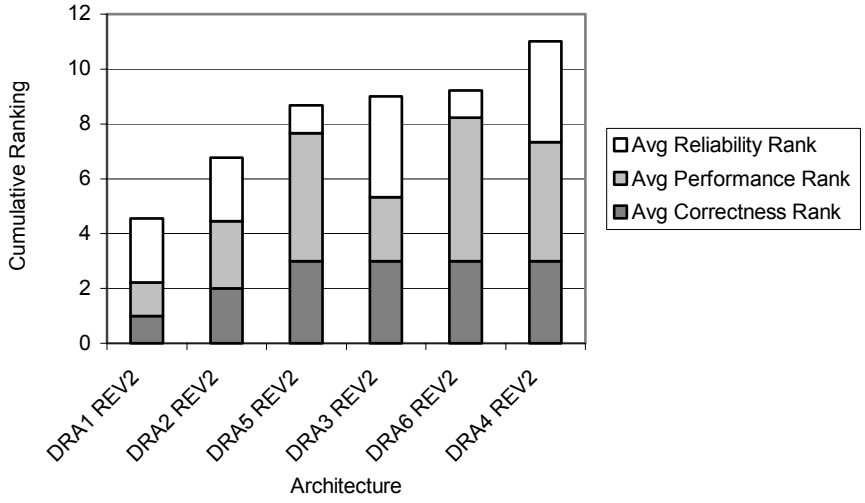


Figure 90- Cumulative Rankings for REV2 DRAs

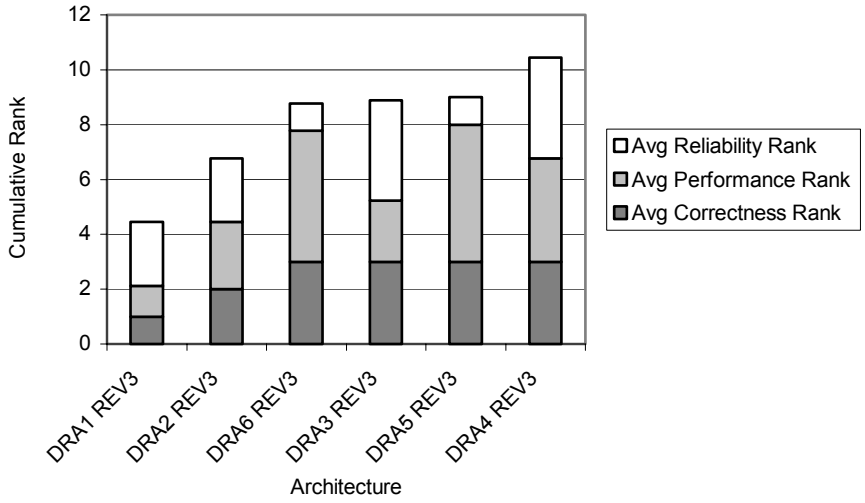


Figure 91 - Cumulative Rankings for REV3 DRAs

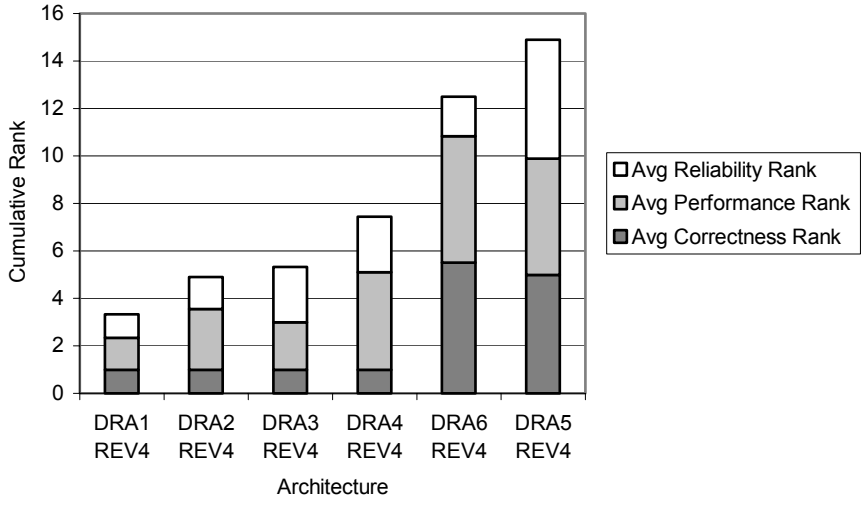


Figure 92 - Cumulative Rankings for REV4 DRAs

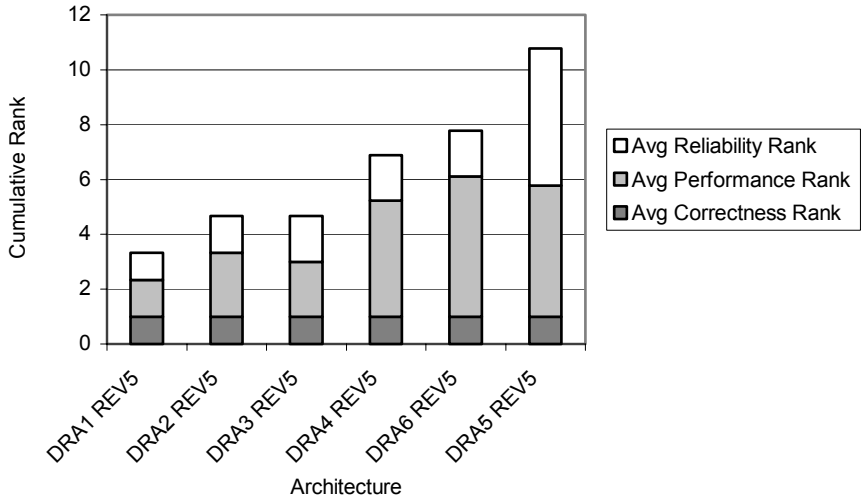


Figure 93 - Cumulative Rankings for REV5 DRAs

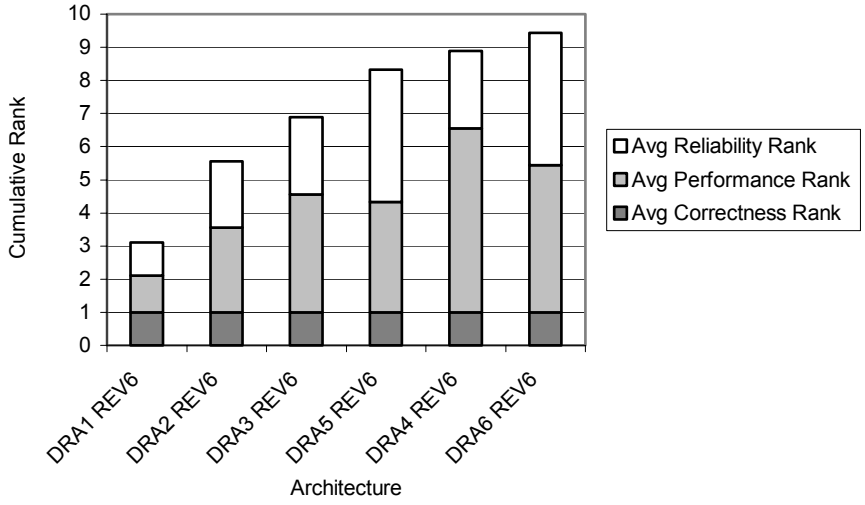


Figure 94 - Cumulative Rankings for REV6 DRAs

An analysis of correlation was performed to determine if the six DRA versions were ranked similarly across requirements revision levels (for example, if DRA1 REV0 was ranked high, was DRA1 REV1 also ranked high?). The Kendall correlation matrix for the seven requirements revision levels is shown in Table 51. The *p-values* are shown in Table 52, and the $R^2$ values are shown in Table 53.

| | REV0 | REV1 | REV2 | REV3 | REV4 | REV5 | REV6 |
|---|---|---|---|---|---|---|---|
| REV0 | 1.0000 | | | | | | |
| REV1 | 0.6900 | 1.0000 | | | | | |
| REV2 | 0.8700 | 0.6900 | 1.0000 | | | | |
| REV3 | 0.7300 | 0.9700 | 0.6000 | 1.0000 | | | |
| REV4 | 0.6000 | 0.5500 | 0.4700 | 0.6000 | 1.0000 | | |
| REV5 | 0.5500 | 0.5000 | 0.4100 | 0.5500 | 0.9700 | 1.0000 | |
| REV6 | 0.8700 | 0.5500 | 0.7300 | 0.6000 | 0.7300 | 0.6900 | 1.0000 |

Table 51 - Correlation Matrix for DRA Version Rankings

| | REV0 | REV1 | REV2 | REV3 | REV4 | REV5 | REV6 |
|---|---|---|---|---|---|---|---|
| REV0 | | | | | | | |
| REV1 | 0.0556 | | | | | | |
| REV2 | 0.0167 | 0.0556 | | | | | |
| REV3 | 0.0556 | 0.0028 | 0.1361 | | | | |
| REV4 | 0.1361 | 0.1361 | 0.2722 | 0.1361 | | | |
| REV5 | 0.1361 | 0.2722 | 0.2722 | 0.1361 | 0.0028 | | |
| REV6 | 0.0167 | 0.1361 | 0.0556 | 0.1361 | 0.0556 | 0.0556 | |

Table 52 - *p-values* for DRA Version Rankings

| | REV0 | REV1 | REV2 | REV3 | REV4 | REV5 | REV6 |
|---|---|---|---|---|---|---|---|
| REV0 | | | | | | | |
| REV1 | 0.7300 | | | | | | |
| REV2 | 0.9200 | 0.9700 | | | | | |
| REV3 | 0.0556 | 0.9700 | 0.9800 | | | | |
| REV4 | 0.2000 | 0.3700 | 0.2400 | 0.3000 | | | |
| REV5 | 0.2800 | 0.4400 | 0.3200 | 0.4000 | 0.9300 | | |
| REV6 | 0.8000 | 0.8400 | 0.8400 | 0.8500 | 0.5800 | 0.5900 | |

Table 53 - $R^2$ Values for DRA Version Rankings

## EXPERIMENT 4: EFFECTS OF ACROSS-ARCHITECTURE EVOLUTION

Details of the experimental setup and conclusions for this experiment are presented in Section 5.8.3.

Results are presented below in graphs depicting rankings for each architecture group from Table 33. In these graphs, the architectures are ordered as DRA, AA, and IA rather than in rank order. This ordering is useful in showing positive or negative impacts of requirements evolution across models.

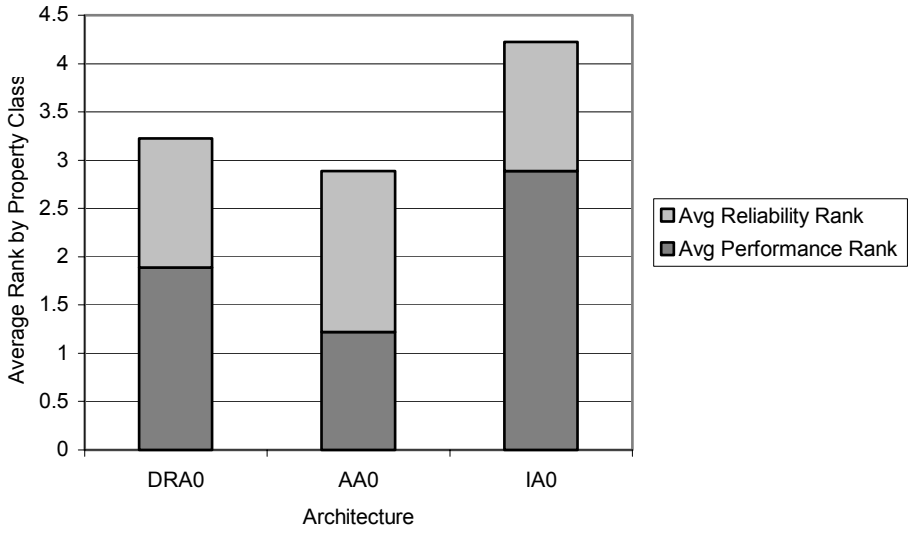


Figure 95 - Rankings for Family 1

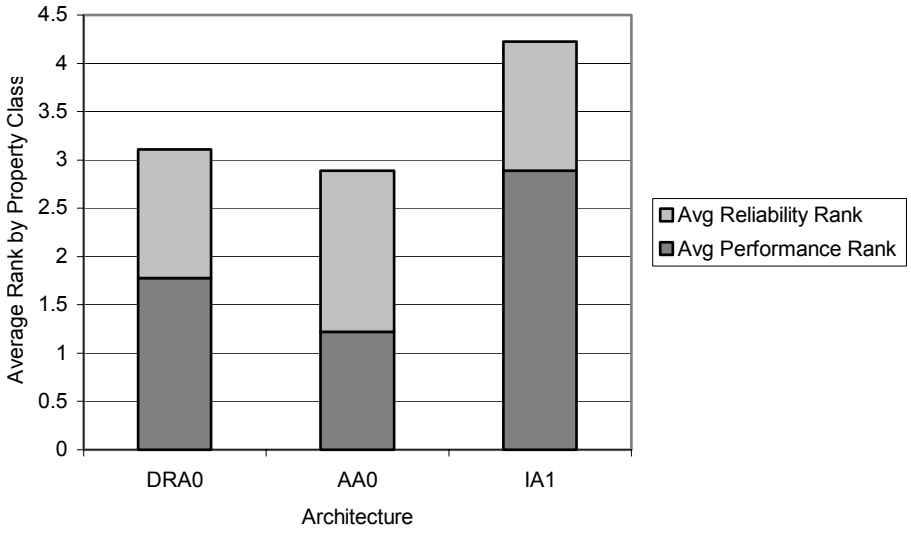


Figure 96 - Rankings for Family 2

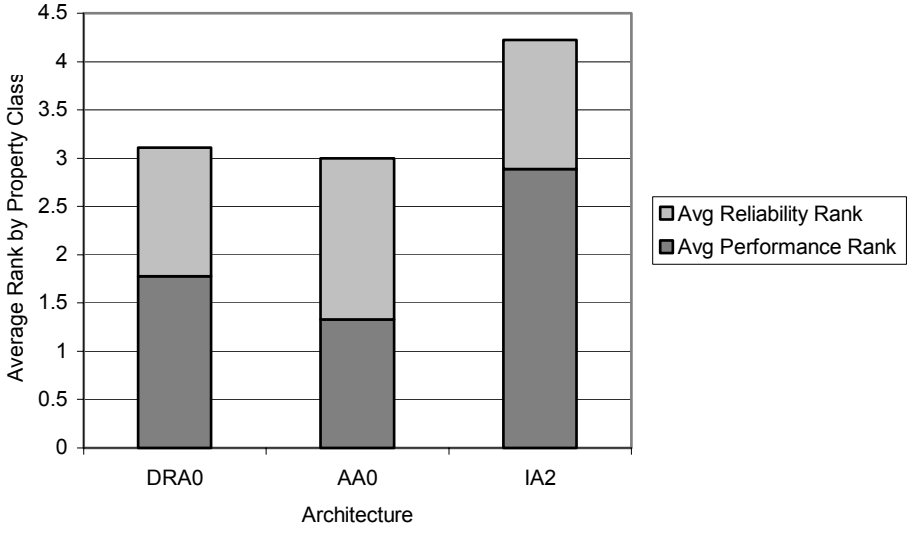


Figure 97 - Rankings for Family 3

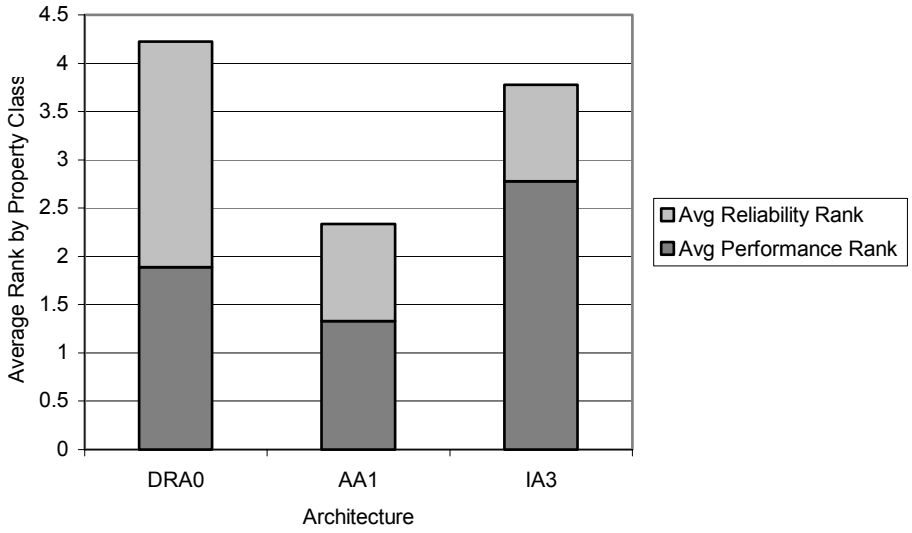


Figure 98 - Rankings for Family 4

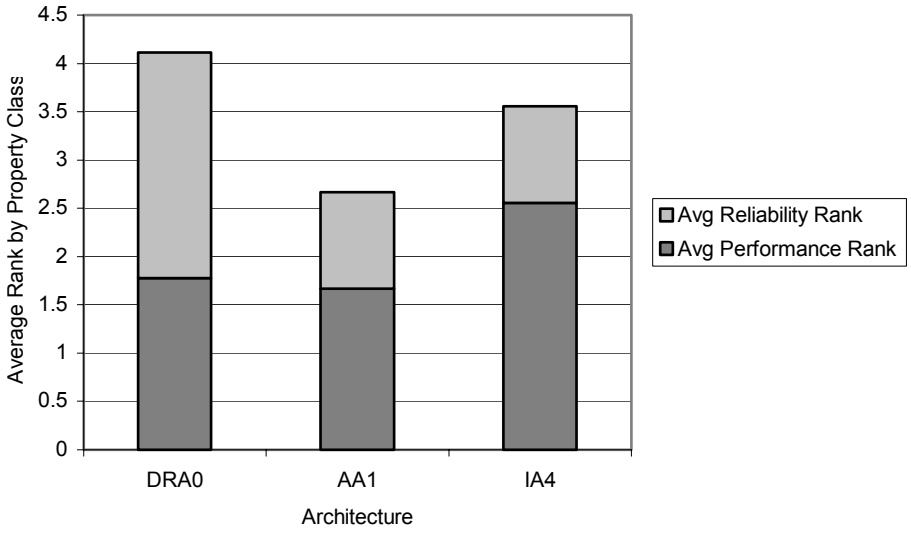


Figure 99 - Rankings for Family 5

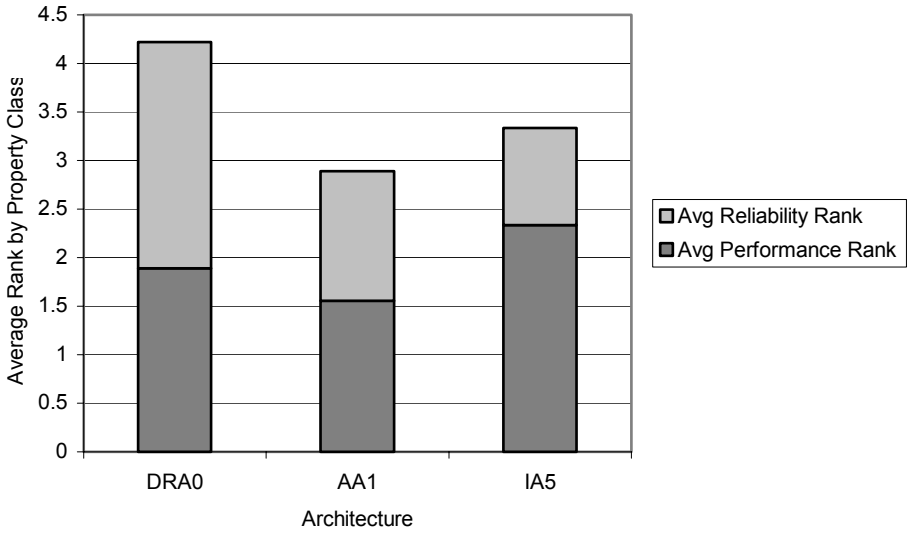


Figure 100 - Rankings for Family 6

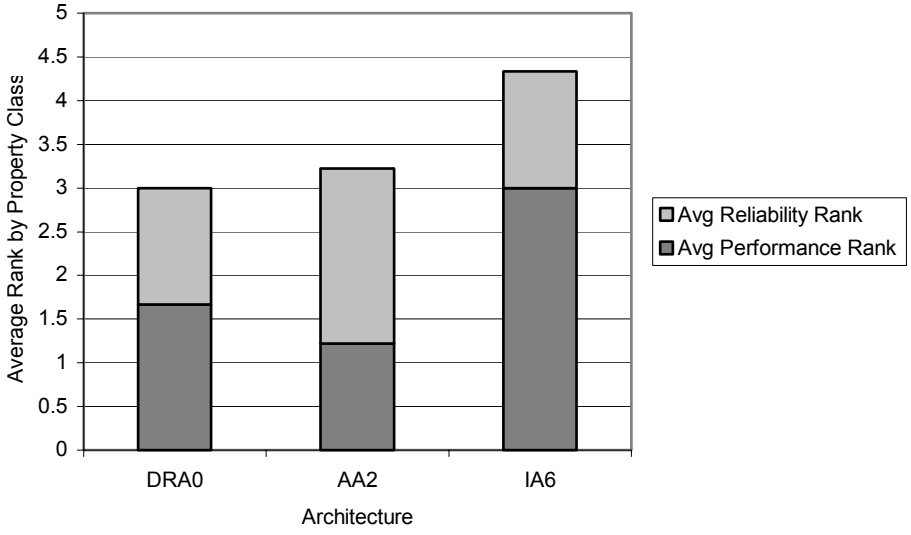


Figure 101 - Rankings for Family 7

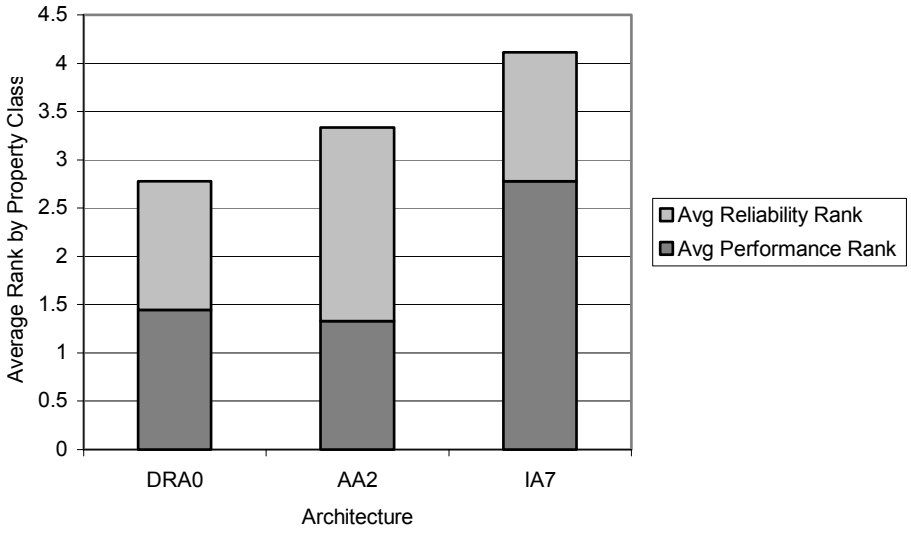


Figure 102 - Rankings for Family 8

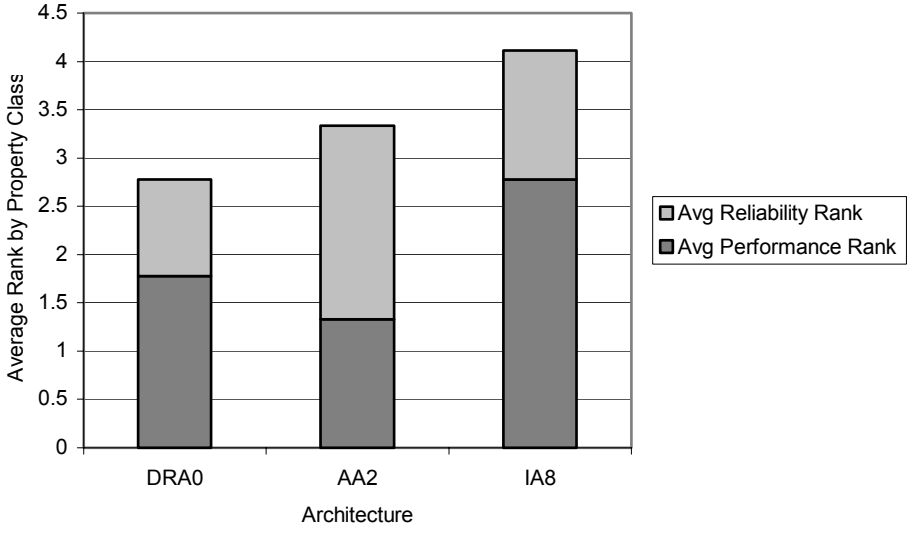


Figure 103 - Rankings for Family 9

# Bibliography


1. Abowd, G., et al., *Recommended Best Industrial Practice for Software Architecture Evaluation*. 1997, Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA.

2. Adve, K.S., et al., *POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems.* IEEE Transactions on Software Engineering, 2000. **26**(11): p. 1027-1048.

3. Andolfi, F., et al. *Deriving Performance Models of Software Architecture from Message Sequence Charts*. in *2nd International Workshop on Software Performance*. 2000.

4. Aquilana, F., S. Balsamo, and P. Inverardi, *Performance Analysis at the Software Architectural Design Level.* Performance Evaluation, 2001. **45**(2-3).

5. Balsamo, S., M. Bernardo, and M. Simeoni. *Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis*. in *3rd International Workshop on Software and Performance*. 2002.

6. Balsamo, S., P. Inverardi, and C. Mangano. *An Approach to Performance Evaluation of Software Architectures*. in *Workshop on Software and Performance*. 1998.

7. Balzer, R., *Instrumenting, Monitoring, & Debugging Software Architectures*. 1995.

8. Barbacci, M., et al., *Quality Attributes*. 1995, Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA.

9. Barber, K.S., et al. *Reliability Estimation Techniques for Domain Reference Architectures*. in *14th International Conference on Software & Systems Engineering and their Applications (ICSSEA 2001)*. 2001. Paris, France.

10. Barber, K.S., et al., *Requirements Evolution and Reuse Using the Systems Engineering Process Activities (SEPA).* Australian Journal of Information Systems (AJIS) - Special Issue on Requirements Engineering, 1999. **7**(1): p. 75-97.

11. Barber, K.S., T.J. Graser, and J. Holt. *Evaluating Dynamic Correctness Properties of Domain Reference Architectures Using a Combination of Simulation and Model Checking*. in *13th International Conference in Software Engineering and Knowledge Engineering (SEKE 2001)*. 2001. Buenos Aires, Argentina.

12. Barber, K.S., T.J. Graser, and J. Holt. *Evolution of Requirements and Architectures: An Empirical-based Analysis*. in *1st International Workshop on Model-based Requirements Engineering (MBRE'01)*. 2001. San Diego, CA.

13. Barber, K.S., T.J. Graser, and J. Holt. *A Multi-Level Software Architecture Metamodel to Support the Capture and Evaluation of Stakeholder Concerns*. in *5th World Multi-conference on Systematics, Cybernatics and Informatics (SCI 2001)*. 2001. Orlando, Florida.

14. Barber, K.S., T.J. Graser, and J. Holt, *Providing Early Feedback in the Development Cycle through Automated Application of Model Checking to Software Architectures*, in *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*. 2001, IEEE: San Diego, California.

15. Barber, K.S., T.J. Graser, and J. Holt. *Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation*. in *17th IEEE International Conference on Automated Software Engineering 2002 (ASE2002)*. 2002.

16. Barber, K.S., et al. *Representing Domain Reference Architectures by Extending the UML Metamodel*. in *12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*. 2000. Chicago, Illinois.

17. Barber, K.S. and J. Holt, *Software Architecture Correctness.* IEEE Software, 2001. **18**(8): p. 64-65.

18. Barber, K.S., J. Holt, and G. Baker. *Performance Evaluation of Domain Reference Architectures*. in *14th International Conference in Software Engineering and Knowledge Engineering (SEKE 2002)*. 2002.

19. Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*. SEI Series in Software Engineering. 1998, Reading, Mass.: Addison Wesley.

20. Batory, D., L. Coglianese, and M. Goodwin, *Creating Reference Architectures: An Example from Avionics.* ACM SIGSOFT Software engineering Notes, 1995. **20**(SI): p. 27-37.

21. Batory, D., et al., *Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study.* ACM Transactions on Software Engineering and Methodology, 2002. **11**(2): p. 191-214.

22. Batory, D. and S. O'Malley, *The Design and Implementation of Hierarchical Software Systems with Reusable Components.* ACM Transactions on Software Engineering and Methodology, 1992. **1**(4): p. 355-398.

23. Bernardo, M., P. Ciancarini, and L. Donatiello. *AEMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures*. in *2nd International Workshop on Software and Performance (WOSP 2000)*. 2000.

24. Boehm, B. and H. In, *Aids for Identifying Conflicts Among Quality Requirements.* IEEE Software, 1996(March 1996).

25. Bosch, J., *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. 2000, Harlow, England: Addison-Wesley.

26. Bosch, J. and P. Molin. *Software Architecture Design: Evaluation and Transformation*. in *IEEE Conference and Workshop on Engineering of Complex Computer Systems*. 1999.

27. Bose, P. *Automated Translation of UML Models of Architectures for Verification and Simulation Using Spin*. in *IEEE International Conference on Automated Software Engineering*. 1999.

28. Bose, P. *Scenario-Driven Analysis of Component-Based Software Architecture Models*. in *IFIP WICSA*. 1999.

29. Browne, J.C. and A. Dube, *Compositional Development of Performance Models in POEMS.* The International Journal of High Performance Computing Applications, 2000. **14**(4): p. 283-291.

30. Browne, J.C. and D. Neuse. *Integration of Performance Engineering Into a New Paradigm for Systems Engineering*. in *High-Level Electronic System Design Conference*. 1997.

31. Carriere, S.J., R. Kazman, and S.G. Woods. *Assessing and Maintaining Architectural Quality*. in *3rd European Conference on Software Maintenance and Reengineering*. 1999.

32. Castaldi, M., P. Inverardi, and S. Afsharian. *A Case Study in Performance, Modificability, and Extensibility Analysis of a Telecommunications System Software Architecture*. in *10th IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '02)*. 2002.

33. Chen, M.-H., M.-H. Tang, and W.-L. Wang. *Effect of Architecture Configuration on Software Reliability and Performance Estimation*. in *IEEE Workshop on Application Specific Software Engineering Technologies (ASSET-98)*. 1998.

34. Cheung, S., D. Giannakopoulou, and J. Kramer. *Verification of Liveness Properties Using Compositional Reachability Analysis*. in *ESEC/FSE 97*. 1997. Zurich, Switzerland.

35. Cheung, S.C. and J. Kramer. *Checking Subsystem Safety Properties in Compositional Reachability Analysis*. in *18th International Conference on Software Engineering*. 1996. Berlin, Germany.

36. Choi, H. and K. Yeom. *An Approach to Software Architecture Evaluation with the 4+1 View Model of Architecture*. in *9th Asia-Pacific Software Engineering Conference*. 2002.

37. Chung, L., B.A. Nixon, and E. Yu. *Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design*. in *1st International Workshop on Architectures for Software Systems*. 1995. Seattle, WA.

38. Compare, D., P. Inerardi, and A.L. Wolf, *Uncovering Architectural Mismatch in Component Behavior.* Science of Computer Programming, 1999. **33**(2): p. 101-131.

39. Contini, S., S. Scheer, and M. Wilikens. *Sensitivity Analysis for System Design Improvement*. in *International Conference on Dependable Systems and Networks*. 2000.

40. Cortellessa, V., H. Singh, and B. Cukic. *Early Reliability Assessment of UML Based Software Models*. in *3rd International Workshop on Software and Performance (WOSP 2002)*. 2002.

41. Cucik, B. *Combining Testing and Correctness Verification in Software Reliability Assessment*. in *High-Assurance Systems Engineering Workshop*. 1997.

42. Dias, M.S. and M.E.R. Vieira. *Software Architecture Analysis Based on Statechart Semantics*. in *10th International Workshop on Software Specification and Design*. 2000.

43. Dobrica, L. and E. Niemela, *A Survey on Software Architecture Analysis Methods.* IEEE Transaction on Software Engineering, 2002. **28**(7): p. 638-653.

44. Duval, G. and T. Cattel. *Developing Safe and Distributed Applications with an Architectural Environment*. in *Technology of Object-Oriented Languages and Systems*. 1999.

45. Egyed, A., P. Gruenbacher, and N. Medvidovic, *Refinement and Evolution Issues in Bridging Requirements and Architectures*. 2000, University of Southern California: Los Angeles, CA.

46. Egyed, A. and D. Wile. *Statechart Simulator for Modeling Architectural Dynamics*. in *IEEE/IFIP Working Conference on Software Architecture*. 2001.

47. Fishwick, P.A., *Simulation Model Design and Execution: Building Digital Worlds*. 1995: Prentice Hall.

48. Fukuzawa, K. and M. Saeki. *Evaluating Software Architectures by Coloured Petri Nets*. in *14th International Conference on Software Engineering and Knowledge Engineering*. 2002.

49. Gannod, G.C. and R.R. Lutz. *An Approach to Architectural Analysis of Product Lines*. in *International Conference on Software Engineering*. 2000.

50. Gnesi, S., et al. *An Automatic SPIN Validation of a Safety Critical Railway System*. in *International Conference on Dependable Systems and Networks*. 2000.

51. Gokhale, S.S., M.R. Lyu, and K.S. Trevedi. *Reliability Simulation of Component-Based Software Systems*. in *9th International Symposium on Reliability Engineering*. 1998.

52. Gokhale, S.S. and K.S. Trivedi. *Reliability Prediction and Sensitivity Analysis Based on Software Architecture*. in *13th International Symposium on Software Reliability Engineering*. 2002.

53. Gokhale, S.S., et al. *An Analytical Approach to Architecture-Based Software Reliability Prediction*. in *IEEE International Computer Performance and Dependability Symposium*. 1998.

54. Gomaa, H. and D.A. Menasce. *Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures*. in *2nd International Workshop on Software and Performance*. 2000.

55. Graser, T., *Reference Architecture Representation Environment (RARE): Systematic Derivation and Evaluation of Domain-Specific, Implementation-Independent Software Architectures*, in *Electrical and Computer Engineering*. 2001, University of Texas at Austin: Austin, TX.

56. Harbison, K., *Scenario-based Engineering Process*. 1997, Center for Advanced Engineering Systems and Automated Research, The University of Texas at Arlington: http://caesar.uta.edu/caesar/process.html.

57. Heitmeyer, C., et al., *Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications.* IEEE Transactions on Software Engineering, 1998. **24**(11): p. 927-948.

58. Heitmeyer, C., J. Kirby, and B. Labaw. *Applying the SCR requirements method to a weapons control panel: An experience report.* in *2nd Workshop on Formal Methods in Software Practice (FMSP'98)*. 1998.

59. Hoeben, F. *Using UML Models for Performance Calculation*. in *2nd International Workshop on Software and Performance (WOSP 2000)*. 2000.

60. Hofmann, H.F. and F. Lehner, *Requirements Engineering as a Success Factor in Software Projects.* IEEE Software, 2001. **18**(4): p. 58-66.

61. Holzman, G.J., *The Model Checker SPIN.* IEEE Transaction on Software Engineering, 1997. **23**(5): p. 279-295.

62. Hsia, P., A. Davis, and D. Kung, *Status Report: Requirements Engineering.* IEEE Software, 1993. **10**(6): p. 75-79.

63. Inverardi, P., et al. *Performance Evaluation of a Software Architecture: A Case Study*. in *9th International Workshop on Software Specification and Design*. 1998.

64. Inverardi, P. and A.L. Wolf, *Formal Specification and Analysis of Software Architecture Using the Chemical Abstract Machine Model.* IEEE Transactions on Software Engineering, 1995. **21**(4): p. 373-386.

65. ITU-TS, *ITU-TS Recommendation Z.120 - Message Sequence Charts (MSC).*. 1996, ITU: Geneva.

66. Jensen, K., *Lectures in Petri Nets II: Applications*, in *Lecture Notes in Computer Science*, W. Reisig and G. Rosenberg, Editors. 1998. p. 237-292.

67. Kazman, R., et al., *Scenario-Based Analysis of Software Architecture.* IEEE Software, 1996. **13**(6).

68. Kazman, R., J. Asundi, and M. Klein. *Quantifying Cost and Benefits of Architectural Decisions*. in *23rd International Conference on Software Engineering*. 2001.

69. Kazman, R., et al. *SAAM: A Method for Analyzing the Properties of Software Architectures*. in *16th International Conference on Software Engineering*. 1994. Sorrento, Italy.

70. Kazman, R., et al. *The Architecture Tradeoff Analysis Method*. in *4th International Conference on Engineering of Complex Computer Systems*. 1998. Monterey, CA.

71. Kim, T., et al. *Software Architecture Analysis Using Dynamic Slicing*. in *AoM/IAoM 1999*. 1999.

72. Kindler, E., *Safety and Liveness Properties: A Survey.* Bulletin of the European Association of Theoretical Computer Science, 1994. **53**: p. 268-272.

73. Kloukinas, C. and V. Issarny. *SPIN-ning Software Architectures: A Method for Exploring Complex Systems*. in *IEEE/IFIP Working Conference on Software Architecture 2001*. 2001.

74. Kuusela, J., A. Maccari, and J. Xu. *Architectural Modeling In Industry - An Experience Report*. in *International Conference on Software Engineering*. 1998.

75. Kuusela, J. and J. Savolainen. *Requirements Engineering for Product Families*. in *22nd International Conference on Software Engineering (ICSE2000)*. 2000.

76. Lamport, L. and N. Lynch, *Distributed Computing: Models and Methods*, in *Handbook of Theoretical Computer Science*, J.v. Leeuwen, Editor. 1990, Elsevier. p. 1157-1199.

77. Land, R. *Improving Quality Attributes of a Complex System Through Architectural Analysis - A Case Study*. in *8th IEEE International Conference and Workshop on Engineering of Computer-Based Systems*. 2002.

78. Larsen, R.J. and M.L. Marx, *An Introduction to Mathematical Statistics and Its Applications*. 2nd Edition ed. 1986, Englewood Cliffs, New Jersey.

79. Lemos, R.d. *Safety Analysis of an Evolving Architecture*. in *5th IEEE International Symposium on High Assurance Systems Engineering*. 2000.

80. Li, J.J. *Performance Prediction Based on Semi-Formal Software Architectural Description*. in *International Conference on Performance in Computing and Communications*. 1998.

81. Li, J.J. and J.R. Horgan. *Simulation-Trace-Based Component Performance Prediction*. in *33rd Annual Simulation Symposium*. 2000.

82. Li, J.J. and J. Micallef. *Automatic Simulation to Predict Software Architecture Reliability*. in *International Symposium on Software Reliability Engineering*. 1997.

83. Lilius, J. and I.P. Paltor. *vUML: a Tool for Verifying UML Models*. in *14th Annual Conference on Automated Software Engineering*. 1999.

84. Lu, J.L., A. Nerode, and V.S. Subrahmanian, *Hybrid Knowledge Bases.* IEEE Transactions on Knowledge and Data Engineering, 1994.

85. Luckham, D. *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events*. in *DIMACS Partial Order Methods Workshop IV*. 1996. Princeton University.

86. Luckham, D.C., et al., *Specification and Analysis of System Architecture Using Rapide.* IEEE Transactions on Software Engineering, 1995. **21**(4): p. 336-355.

87. Luckham, D.C., J. Vera, and S. Meldal, *Three Concepts of System Architecture*. 1995, Stanford University.

88. Lung, C.-H., A. Jalnapurkar, and A. El-Rayess. *Performance-Oriented Software Architecture Engineering: an Experience Report*. in *Workshop on Software Performance (WOSP98)*. 1998. Santa Fe, N.M.

89. Lyu, J., J.-H. Ding, and H. Luh, *Petri nets for performance modeling study of client-server systems.* International Journal of Systems Science, 1998. **29**(6): p. 565-571.

90. Magee, J., J. Kramer, and D. Giannakopoulou. *Analysing the Behaviour of Distributed Software Architectures: a Case Study*. in *5th IEEE Workshop on Future Trends of Distributed Computing*. 1997. Tunisia.

91. Merseguer, J., J. Campos, and E. Mena. *Performance Analysis of Internet based Software Retrieval using Petri Nets*. in *4th ACM International Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems*. 2001.

92. Nakajima, S. and T. Tamai. *Behavioral Analysis of the Enterprise JavaBeans Component Architecture*. in *The 8th International SPIN Workshop on Model Checking of Software*. 2001.

93. Naumovich, G., et al. *Applying Static Analysis to Software Architectures*. in *6th European Software Engineering Conference*. 1997. Zurich, Switzerland.

94. Perry, D.E. and A.L. Wolf, *Foundations for the Study of Software Architecture.* Software Engineering Notes, 1992. **17**(4): p. 40-52.

95. Petriu, D., C. Shousha, and A. Jalnapurkar, *Architecture-based Performance Analysis Applied to a Telecommunication System.* IEEE Transactions on Software Engineering, 2000. **26**(11): p. 1049-1065.

96. Petriu, D. and M. Woodside. *Analysing Software Requirements Specifications for Performance*. in *3rd International Workshop on Software and Performance*. 2002.

97. Porkess, R., *The Harper Collins Dictionary of Statistics*. 1991, New York, N.Y.: Harper Collins.

98. Rational Partners (Rational et al.), *Object Constraint Language Specification*. 1997, Object Management Group.

99. Rational Partners (Rational et al.), *OMG UML Specification v1.3*. 1999, Object Management Group.

100. Rosa, N.S., G.R.R. Justo, and P.R.F. Cunha. *A Framework for Building Non-Functional Software Architectures*. in *16th ACM Symposium on Applied Computing (SAC2001)*. 2001.

101. Saridakis, T. and V. Issarney. *Developing Dependable Systems Using Software Architecture*. in *1st Working IFIP Conference on Software Architecture*. 1999. San Antonio, TX.

102. Schneider, F., et al. *Validating Requirements for Fault Tolerant Systems using Model Checking*. in *3rd International Conference on Requirements Engineering*. 1998.

103. Sharareh, A., G. Matteo, and T. Gianluca. *Quantitative Analysis for Telecom/Datacom Software Architecture*. in *3rd International Workshop on Software and Performance*. 2002.

104. Sommerville, I., *Software Engineering*. 4th ed. 1992, Wokingham, England: Addison-Wesley.

105. Spitznagel, B. and D. Garlan. *Architecture-based Performance Analysis*. in *Conference on Software Engineering and Knowledge Engineering*. 1998.

106. Tsai, J. and K. Xu, *An Empirical Evaluation of Deadlock Detection in Software Architecture Specifications.* Annals of Software Engineering, 1999. **7**: p. 95-126.

107. Vieira, M.E.R., M.S. Dias, and D.J. Richardson. *Analyzing Software Architectures with Argus-I*. in *22nd International Conference on Software Engineering*. 2000.

108. Wang, W.-L., M.-H. Tang, and M.-H. Chen. *Software Architecture Analysis - A Case Study*. in *Computer Software and Applications Conference (COMPSAC)*. 1999.

109. Wang, W.-l., Y. Wu, and M.-H. Chen. *An Architecture-Based Software Reliability Model*. in *1999 Pacific Rim International Symposium on Dependable Computing*. 1999.

110. Wieringa, R. and E. Dubois, *Integrating Semi-Formal and Formal Software Specification Techniques.* Information Systems, 1998. **23**(3/4): p. 159-178.

111. Williams, L.G. and C.U. Smith. *Performance Evaluation of Software Architectures*. in *Workshop on Software and Performance*. 1998. Santa Fe, N.M.

112. Williams, L.G. and C.U. Smith. *PASA: A Method for the Performance Assessment of Software Architectures*. in *3rd International Conference on Software and Performance*. 2002.

113. Woodside, C.M. *Software Resource Architecture and Performance Evaluation of Software Architectures*. in *34th Hawaii International Conference on System Sciences*. 2001.

114. Woodside, M., D. Petriu, and K. Siddiqui. *Performance-related Completions for Software Specification*. in *International Conference on Software Engineering*. 2002.

115. Xie, F., V. Levin, and J.C. Browne. *Model Checking for an Executable Subset of UML*. in *16th International Conference on Automated Software Engineering*. 2001.

116. Xu, J. and J. Kuusela. *Modeling Execution Architecture of Software System Using Colored Petri Nets*. in *1st International Workshop on Software and Performance*. 1998.

117. Yacoub, S., B. Cukic, and H. Ammar. *Scenario-Based Reliability Analysis of Component-Based Software*. in *10th International Symposium on Software Reliability Engineering*. 1999. Boca Raton, FL.

118. Yacoub, S.M. and H.H. Ammar, *A Methodology for Architecture-Level Reliability Risk Analysis.* IEEE Transactions on Software Engineering, 2002. **28**(6): p. 529-547.

119. Yu, H., et al. *A Formal Method for Analyzing Software Architecture Models in SAM*. in *25th Annual International Computer Science and Applications Conference (COMPSAC '02)*. 2002.

# Vita

James Carrell Holt was born in 1959 to James and Gayle Holt in Corpus Christi, Texas. During his childhood he attended school in Texas and Germany, and lived briefly in Alaska. In 1991 he earned a B.S. in Computer Science from St. Edward's University in Austin, Texas. He graduated Summa Cum Laude, and was selected by the Computer Science faculty as the 1990-1991 Outstanding Computer Science Student. He began graduate studies at Southwest Texas State University in San Marcos, Texas, where he was awarded a scholarship for Academic Excellence in Graduate Studies. He graduated with a Master of Science in Computer Science in December 1996. After taking one year off from his studies, he began his doctoral research in software engineering in 1998 at the University of Texas at Austin. Throughout his academic career, James has been employed full time and has had his education supported by Motorola, Inc., of Schaumburg Illinois. He completed 20 years of service with Motorola in February, 2003, and is currently a Distinguished Member of the Technical Staff working on advanced design verification tools for integrated circuits. He has received two United States patents for inventions he developed during the course of his employment with Motorola. He is currently a member of the Institute of Electrical and Electronics Engineers (IEEE).

Permanent address: 8005 Henry Kinney Row
Austin, TX 78749

This dissertation was typed by the author.